Adrian Singer

Entwicklung und Implementierung einer 3D-Visualisierungsbibliothek und der Graphischen Benutzerschnittstelle einer Anwendungssoftware für die Simulation und Demonstration der Kommunikation in drahtlosen Sensornetzwerken

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Informationstechnik & Elektrotechnik

Mittweida, 2010

Adrian Singer

Entwicklung und Implementierung einer 3D-Visualisierungsbibliothek und der Graphischen Benutzerschnittstelle einer Anwendungssoftware für die Simulation und Demonstration der Kommunikation in drahtlosen Sensornetzwerken

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Informationstechnik & Elektrotechnik

Mittweida, 2010

Erstprüfer: Prof. Dr.-Ing. Volker Delport
Zweitprüfer: Prof. Dr.-Ing. Wilfried Schubert

Die vorliegende Arbeit wurde verteidigt am:

Vorwort und Danksagung

Die Anfertigung einer Diplomarbeit ist ein langwieriger Prozess, der nicht ohne Hilfe bewältigt werden kann. Hier möchte ich all jenen meinen Dank aussprechen, die mich auf diesem Weg begleitet und unterstützt haben. Darüber hinaus möchte ich betonen, dass es mir eine besondere Freude war, einen Beitrag zu diesem sehr interessanten und umfassenden Forschungsprojekt leisten zu können.

An erster Stelle möchte ich mich bei meinem Erstbetreuer Herrn Prof. Dr.-Ing. Volker Delport dafür bedanken, dass er mir dieses Thema anvertraute und mir somit die Möglichkeit bot, diese Arbeit anzufertigen.

Ebenso dankbar bin ich für die Unterstützung durch meinen Zweitbetreuer Prof. Dr.-Ing. Wilfried Schubert und seine zahlreichen Ratschläge.

Für das gute Arbeitsklima in der Forschungsgruppe möchte ich mich bei all meinen Kollegen bedanken.

Ein ganz besonderer Dank gilt meiner Familie, die mir stets den Freiraum für die Bearbeitung dieser Arbeit gegeben hat und mich auf dem Weg zu meinem Abschluss stets unterstützte.

Vielen Dank.

Bibliografische Beschreibung

Singer, Adrian:
Entwicklung und Implementierung einer 3D-Visualisierungsbibliothek und der Graphischen Benutzerschnittstelle einer Anwendungssoftware für die Simulation und Demonstration der Kommunikation in drahtlosen Sensornetzwerken
- 2010 - 90 S
Mittweida, Hochschule Mittweida (FH), Fakultät Informationstechnik & Elektrotechnik, Diplomarbeit, 2010

Referat

Im Rahmen eines Forschungsprojektes werden drahtlose Sensornetze untersucht. Parallel findet die Entwicklung einer Simulations- und Demonstrationssoftware für diese Netzwerke statt. Das erste Ziel der Diplomarbeit ist die Entwicklung und Realisierung der gesamten Anwendungsoberfläche des Simulators/Demonstrators. Des weiteren ist es das Ziel dieser Arbeit, eine Klassenbibliothek zu entwickeln, mit der es möglich ist, drahtlose Sensornetze in einer dreidimensionalen Umgebung darzustellen. Diese Visualisierungsbibliothek wird auf der Basis der .NET-Programmiersprache C# und der Bibliothek OpenGL realisiert. Hinzu kommt die Implementierung der Klassenbibliothek in einer Anwendung. Die 3D-Bibliothek bildet nicht den gesamten Funktionsumfang von OpenGL ab, da sie auf die Anforderungen der Simulationsanwendung angepasst ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
C / C++	Die Programmiersprache C beziehungsweise C++
C#	die Programmiersprache[C-Sharp]
CPU	Central Processing Unit
DLL	Dynamic Link Library
PC	Personal Computer
GLU	OpenGL Utility Library
GPU	*Graphics Processing Unit*
GUI	Graphical User Interface
IDE	Integrated Development Environment
MDI	Multiple Document Interface
OpenAL	Open Audio Library
OpenCL	Open Computing Library
OpenGL	Open Graphics Library
UML	Unified Modeling Language
WLAN	Wireless Local Area Network
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

1 Einleitung

1.1 Hintergrund und Motivation

Drahtlose Netzwerke besitzen ein ungleich höheres Potential, als ihre drahtgebundenen Vorläufer. Sie sind an keine feste Stromquelle gebunden und können somit theoretisch auch in schwer zugänglichen Gebieten eingesetzt werden. Derzeit sind drahtlose Netze aber noch nicht völlig flexibel einsetzbar. Bei WLAN ist beispielsweise immer ein fest installierter Access Point nötig. Hinzu kommt, dass alle Endgeräte eine direkte Funkverbindung zu diesem haben müssen.

Eine andere Netztopologie, das so genannte Adhoc-Netzwerk, kann völlig auf eine festgelegte Infrastruktur verzichten. Diese Netzwerke können sich auch ohne Access Point spontan auf- und abbauen, stehen jedoch in ihrer Entwicklung noch am Anfang. Werden solche Netze als Sensornetzwerk eingesetzt, dann ergibt sich eine Vielzahl neuer Einsatzmöglichkeiten. Einige Beispiele sind die Temperatur-überwachung großer Areale, Mess- und Kontrollsysteme in vielen Bereichen der Wirtschaft und Industrie oder zur Überwachung von Patienten in der Medizin.

Die Kommunikation der einzelnen Netzwerkknoten in solch einem Netzwerk kann auf direktem Wege erfolgen. Besteht jedoch keine direkte Funkverbindung zwischen Sender und Empfänger, so können dazwischen liegende Teilnehmer die Daten weiterleiten. Für diese Aufgabe werden spezielle Routingprotokolle bereit gestellt. Der Weg, den die Datenpakete durch das Netzwerk nehmen, soll dabei möglichst effizient sein. In einigen Fällen findet eine Gruppierung des gesamten Netzwerkes in mehrere Cluster statt. Der Vorteil von Clustering-basierten Routingprotokollen ist die Entlastung der Endgeräte innerhalb der Gruppe. Diese

können mit verringerter Sendeleistung ihre Daten, an den nahe liegenden „Cluster Head“ übermitteln. Dem entgegen steht die Belastung des Knotens, der als Cluster Head fungiert. Er muss die Vielzahl der Daten aufnehmen, gegebenenfalls aufbereiten und weiter an eine Basisstation senden. Die Abbildung 1-1 zeigt die schematischen Funkverbindungen eines clusterbasierten Sensornetzwerkes.

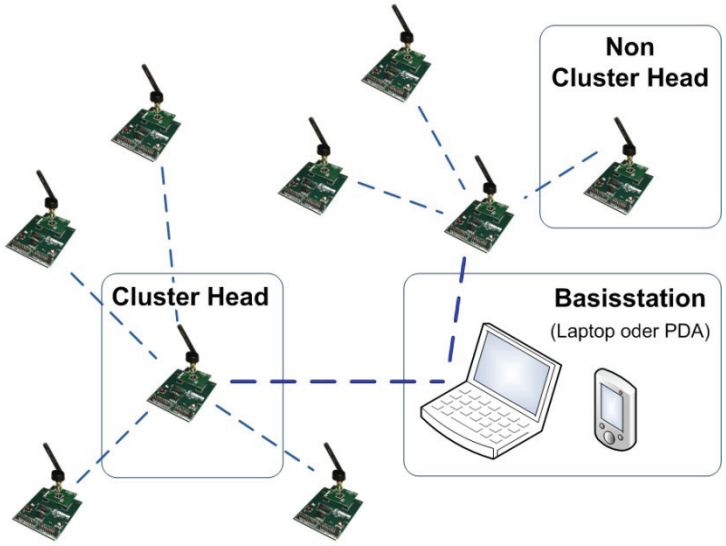


Abbildung 1-1 clustering-basiertes Sensornetzwerk

Der Energieverbrauch, der durch die drahtlose Kommunikation zwischen den Sensorknoten entsteht, beeinflusst die Lebensdauer des gesamten Sensornetzes. Ziel des Forschungsprojektes ist es, die eingesetzten Routingprotokolle, Clustering-Verfahren und die Energieverwaltung bei jedem Netzwerkknoten entscheidend zu verbessern. Daran forschen und entwickeln Funk- und Kommunikationstechniker, sowie Informatiker gleichermaßen. Damit die Forschungsergebnisse ausgewertet und zukünftige Sensornetze intelligent geplant und aufgebaut werden können, benötigt man eine Simulations- und Demonstrationssoftware. Die Diplomarbeit von Toni Dirk Großmann [Großmann 09] befasste sich mit dem Systementwurf einer solchen Anwendung und bildet die Grundlage dieser Arbeit.

Die Programmierung des Simulators/Demonstrators ist somit entscheidend für die weitere Entwicklung hochmoderner und intelligenter Netzwerke. Das Programm muss Vorgänge, wie Clustering, Routing oder die Visualisierung der Datenübertragung möglichst genau widerspiegeln.

1.2 Ziele der Arbeit

Trotz des einen übergeordneten Forschungsthemas, welches sich mit der Optimierung von Sensornetzen befasst, gibt es für diese Arbeit eine klare Zweiteilung der Aufgaben. Das erste große Aufgabenfeld befasst sich mit der gesamten Softwareoberfläche des Simulators/Demonstrators. Diese soll mit dem .Net Framework von Microsoft entworfen und implementiert werden. Zur Vereinfachung des Simulationsmodells, ist es zunächst angedacht, eine schematische 2D-Ansicht eines Sensornetzes zu realisieren. Der Simulationsvorgang soll konfiguriert, gesteuert, angezeigt und ausgewertet werden können. Die Konfiguration eines Netzwerkes soll über dynamisch aufgebaute Einstellungsfenster möglich gemacht werden. Die Auswertung des Simulationsablaufs soll mittels weiterer Diagramme visualisiert werden können. Es wird gefordert, dass die grafische Benutzeroberfläche hohen Ansprüchen in Bezug auf die Erweiterbarkeit und die Gebrauchstauglichkeit genügt.

Damit ein realistisches Abbild eines Sensornetzwerkes entsteht, soll im zweiten großen Teil dieser Arbeit eine Visualisierungsbibliothek für dreidimensionale Inhalte geschaffen werden. Um eine 3D-Umgebung darstellen zu können, muss die entsprechende Klassenbibliothek realisiert und implementiert werden. Diese DLL stellt dafür alle wichtigen Funktionen aus der OpenGL-Bibliothek bereit. Es wird nicht gefordert, dass alle OpenGL spezifischen Anweisungen für die Eigenentwicklung übersetzt werden. Die C# Spezifikation der OpenGL-Bibliothek soll flexibel und leicht verständlich gestaltet werden. Die damit entstandene 3D-

Grafikschnittstelle für .Net Applikationen soll schließlich in einer Anwendung, beispielsweise im Simulator/Demonstrator, integriert und verwendet werden. Das Erstellen einer 3D-Software, die auf dem .Net Framework basiert, soll auch als Ausblick für künftige Weiterentwicklungen dienen.

1.3 Kapitelaufbau

Anhand der Ziele dieser Arbeit lässt sich bereits die Zweiteilung des gesamten Sachverhaltes erkennen. Das zweite Kapitel dieser Arbeit legt die Grundlagen, die zum Verstehen der Oberflächenprogrammierung mit dem .Net Framework notwendig sind. Des weiteren wird eine Einführung in die 3D-Grafik Programmierung gegeben.

Das dritte Kapitel erstreckt sich über die erste große Aufgabenstellung dieser Abschlussarbeit. Zu Beginn wird eine Anforderungsanalyse der Programmoberfläche durchgeführt. Im Anschluss daran werden verschiedene Lösungsmöglichkeiten diskutiert. Die Umsetzung der bevorzugten Variante wird ausführlich beschrieben und mit einem Fazit abgeschlossen. Ein Lösungsansatz, wodurch die Softwareoberfläche die Fähigkeit der Mehrsprachigkeit besitzt, wird ebenfalls erläutert und veranschaulicht.

Das zweite Themengebiet innerhalb dieser Arbeit wird im Kapitel vier ausgearbeitet. Begonnen wird mit einer Einführung in die OpenGL Technologie, worauf die eigens entwickelte Visualisierungsbibliothek aufsetzt. Die Ansprüche, die an diese C# spezifische 3D-Schnittstelle gestellt werden, werden ausführlich heraus gearbeitet. Im Anschluss daran erfolgt eine Beschreibung der umgesetzten Strukturen und wie man diese 3D-Bibliothek anwendet. Im Kapitel fünf werden die erreichten Ziele der geforderten Aufgabenstellung gegenüber gestellt. Es wird ein Ausblick gegeben, wie die Weiterentwicklung der bearbeiteten Teilaufgaben innerhalb des Forschungsprojektes verlaufen könnte.

2 Grundlagen zur Programmiertechnologie

2.1 Programmierung mit C#

Die Programmiersprache C# ist eine Entwicklung des Softwareherstellers Microsoft. Sie gehört zu den objektorientierten Programmiersprachen und ist ein Bestandteil des .NET Frameworks. Eingesetzt wird die Entwicklungsumgebung *Visual Studio 2008*. Mit Hilfe dieser Laufzeitumgebung ist es möglich, Software zu entwickeln, die für unterschiedliche Geräte, beispielsweise PC oder PDA, gedacht ist. Auf anderen Betriebssystemen, wie Linux, laufen .Net Anwendungen nur teilweise. Für den praktischen Teil dieser Arbeit wurde überwiegend C# verwendet. Neben der Sprache C# kommt zur Verwaltung von Datenstrukturen auch XML[1] zum Einsatz. Für diese Scriptsprache wird die weiterführende Literatur [Vonhoegen 09] empfohlen. Weitere diskutierte Technologien der Oberflächen-programmierung, die von Microsoft entwickelt werden, sind XAML [2] oder WPF [3]. Aus Zeitgründen soll hier ebenfalls nur auf das Buch [Huber 08] verwiesen werden.

Standardmäßig wird die dreidimensionale Darstellung mittels OpenGL unter C# nicht unterstützt. Damit entsteht die Notwendigkeit, Funktionen aus der OpenGL Bibliothek in C# zu importieren. Dies geschieht über eine Wrapper-DLL[4], welche die OpenGL Befehle in die Sprache C# übersetzt. Dafür werden einige Klassen zur Strukturierung eingerichtet, um Funktionen zu sortieren oder zu kapseln. Die so entstandene Klassenbibliothek kann in jedem beliebigen C#-Projekt eingebunden und genutzt werden. Die genaue Erläuterung über die Funktionsweise von

[1] Extensible Markup Language
[2] Extensible Application Markup Language
[3] Windows Presentation Foundation
[4] Wrapper-Dynamic Link Library, deutsch: umschließende dynamische verlinkte Bibliothek

OpenGL und wie die Entwicklung der Wrapper-DLL gestaltet wurde, folgt im Kapitel 4 "Die Visualisierungsbibliothek für OpenGL".

2.2 Die graphische Benutzeroberfläche

Die Programmierung einer Softwareoberfläche mit den gängigen Steuerelementen wie Buttons, Textfelder, Dialogen und Tabellen wird durch *Visual Studio* bestens unterstützt. Das Positionieren von Komponenten und das Anlegen von Ereignissen lässt sich über einen grafischen Editor leicht erreichen. Es ist hingegen sehr viel schwerer, die Oberfläche einer Applikation intelligent zu planen und zu realisieren, sodass deren Handhabung leicht fällt.

In der Softwarebranche fällt häufig der Begriff "Usability", was von der Allgemeinheit recht schnell mit „Benutzerfreundlichkeit“ gleich gesetzt wird. Diese Übersetzung ist nicht ganz korrekt. Sie kann außerdem von Anwender zu Anwender ganz unterschiedlich bewertet werden. Viel genauer trifft der deutsche Begriff „Gebrauchstauglichkeit“ zu. Damit ist es möglich zu bewerten, ob eine Softwareoberfläche für den alltäglichen Gebrauch geeignet ist, oder nicht.

Es gibt den EN ISO 9241 Standard „Ergonomie der Mensch-System-Interaktion“. [ISO9241 06] In dessen Teil 11 sind drei Leitkriterien festgehalten nach denen die Gebrauchstauglichkeit bemessen werden kann. Es ist demnach zu bewerten, wie effektiv die Software bei der Lösung einer Aufgabe ist, wie effizient die Handhabung ist und wie die Zufriedenheit des Nutzers ausfällt.

Neben diesen Kriterien gibt es auch weitere Richtlinien für ergonomisch designte Software. In seiner Vorlesung "Grundlagen der Softwaretechnik" [Schubert 09] fasst Wilfried Schubert diese wie folgt zusammen:

Aufgabenangemessenheit • geeignete Funktionalität, Minimierung der Interaktionsschrittzahl
Selbstbeschreibungsfähigkeit • Verständlichkeit durch Texte, Hilfen, Rückmeldungen
Steuerbarkeit • sollte durch Benutzer möglich sein
Erwartungskonformität • Konsistenz der Bedienschritte, Quasistandards
Fehlertoleranz • erkannte Fehler verhindern nicht das Benutzerziel, unerkannte Fehler → leichte Korrektur
Individualisierbarkeit • Anpassbarkeit an Benutzer und Arbeitskontext
Lernförderlichkeit • Anleitung des Benutzers, minimale Anlernzeit, Arbeiten mit Metaphern

Tabelle 2-1 Kriterien der Softwareergonomie

Anhand dieser Informationen kann man nun eine Reihe wichtiger Punkte für die Oberflächenprogrammierung des Simulators/Demonstrators ableiten. Diese sind im Kapitel 3.1 "Anforderungsanalyse" festgehalten.

2.3 Grafikprogrammierung

Um genauer verstehen zu können, wie auf einem Monitor Bilder angezeigt werden, soll der Vorgang, beginnend bei der Hardware, bis hin zur Bildschirmausgabe nachvollzogen werden. Auf dem Mainboard eines Computers befindet sich neben der zentralen Recheneinheit, der CPU [5], auch immer ein

[5] Central Processing Unit

Grafikprozessor, der die Daten für die Monitorausgabe berechnet. Diese Onboard-Variante, bei der sich die GPU[6] direkt mit auf der Hauptplatine befindet, hat einige Nachteile. Zum einen besitzt dieser Grafikprozessor keinen eigenen Speicher. Zum anderen sind diese Onboard-Lösungen bei weitem nicht so leistungsfähig, wie eine gesteckte Grafikkarte.

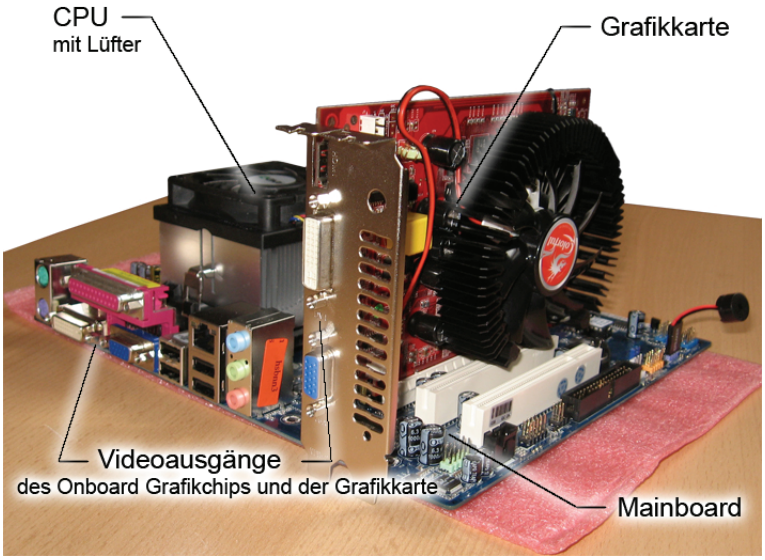


Abbildung 2-1 Mainboard mit gesteckter Grafikkarte

Die Abbildung 2-1 zeigt eine *nVIDIA GeForce 9600GT* Grafikkarte, die über den PCI-Express Anschluss des Mainbords vom Hersteller *ASRock* angesteckt wurde. Wie man sieht, verdeckt der Lüfter der Grafikkarte die meisten Komponenten. Der wichtigste Hardware Bestandteil ist der Grafikchip mit 3D-Beschleuniger. Dieser wird unter anderem für dreidimensionale Berechnungen benötigt. Weiterhin besitzen gesteckte Grafikkarten einen eigenständigen Speicher, den so genannten Bildspeicher oder Video-RAM. In dieser Speichereinheit wird ein digitales Abbild des Monitorbildes abgelegt. Diese Daten werden dann über ein Kabel, analog oder digital, an den Monitor gesendet und so lange wiederholt angezeigt, bis ein neues Bild berechnet wurde.

[6] Graphics Processing Unit

DirectX und OpenGL

Die Programmierung von Grafiken, insbesondere 3D-Grafiken erfordern einige Kenntnisse über die darunter liegenden Schichten. Grundsätzlich gibt es zwei Technologien, die von heutigen Grafikkarten unterstützt werden, um dreidimensionale Szenen darstellen zu können. Eine der beiden nennt sich DirectX und die andere wird mit OpenGL bezeichnet. Beide Programmierschnittstellen werden abhängig von der vorhandenen Hardware unterschiedlich gut unterstützt.

Zum einen gibt es von Microsoft die DirectX API[7]. Dieses Funktionspaket wird neben Windows auch von der Spielekonsole XBox verwendet. Damit wird deutlich, dass DirectX hauptsächlich für Multimedia- und Spieleanwendungen entwickelt wird. Es findet auch großen Zufluss bei Spiele-Entwicklern, da DirectX neben der 3D-Darstellung auch Komponenten für Sound, Netzwerkverbindungen und Controllersteuerung mitliefert.

Die zweite Technologie ist die Open Graphics Library, oder kurz OpenGL. Dies ist eine frei zugängliche Grafikbibliothek, also ein offener Standard, der von vielen Herstellern aus der Soft- und Hardwarebranche entwickelt wird. Dazu gehören unter anderem Apple, ATI, DELL, IBM, Intel, nVIDIA und Sun. Diese und einige weitere Firmen haben sich zum OpenGL Architecture Review Board, dem ARB, zusammengeschlossen. Durch die vielen Interessengruppen entsteht eine Grafikschnittstelle, die unabhängig vom verwendeten Betriebssystem eingesetzt werden kann. Hinzu kommt, dass die meisten semiprofessionellen und professionellen Grafikkarten eine bessere Hardwareunterstützung für OpenGL liefern, als für DirectX.

[7] Application Programming Interface, deutsch: Schnittstelle zur Anwendungsprogrammierung

Für die Ansprüche des Forschungsprojektes ist die Verwendung von OpenGL besser geeignet, da man, mit einer auf OpenGL gestützten Software, nicht zwangsläufig an das Betriebssystem Windows gebunden ist. Bei OpenGL ist eine stete Entwicklung zu erkennen, die auf Grafikkarten von ATI oder nVIDIA bestens implementiert wird. Zudem läuft man mit OpenGL nicht Gefahr, auf kurzfristige Änderungen oder künftige Microsoft Entwicklungen reagieren zu müssen, da deren wichtigste Zielgruppe die Spiele-Industrie ist.

2.4 Grundbegriffe der 3D-Grafik Programmierung

Aufbau von Objekten

Bei beiden Programmierschnittstellen, sowohl bei DirectX, als auch bei OpenGL, ist das Gerüst einer 3D-Grafik nahezu identisch. Dennoch beziehen sich die nun folgenden Grundlagen stets auf die OpenGL-API, da ausschließlich mit dieser Grafikschnittstelle gearbeitet wurde. Wie die Abbildung 2-2 demonstriert, bestehen alle Objekte aus Punkten, den Vertices (plural für Vertex), Linien beziehungsweise Kanten und Flächen, den Polygonen. Die dreidimensionalen Objekte sind dabei keine vollständig ausgefüllten Körper, denn sie besitzen nur eine Oberfläche und sind im Inneren hohl. Ein Polygon kann als Dreieck, Viereck oder als N-Eck auftreten, solang die Innenwinkel der Kanten kleiner als 180° sind. Diese Bestandteile werden dann auf der Grafikkarte entsprechend ihrer räumlichen Position abgespeichert. Aus diesen Vektordaten berechnet der Grafikchip eine Rastergrafik, also ein pixelbasiertes Bild, was schließlich auf dem Monitor ausgegeben wird. Der exemplarische Ablauf, wie man mit Hilfe von OpenGL ein Bild zeichnet, wird im Kapitel 4.2 "Einführung in OpenGL" erklärt.

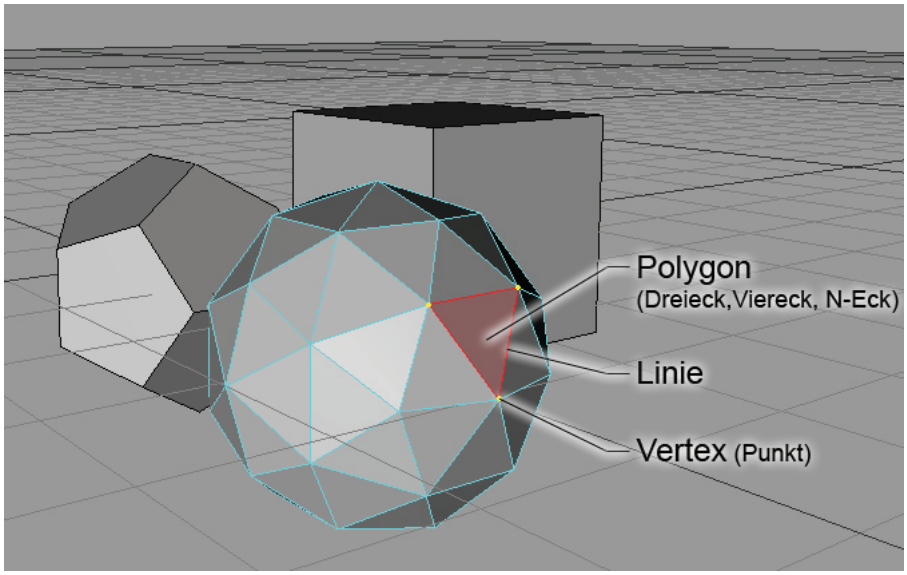


Abbildung 2-2 Aufbau von 3D Objekten

Wie bereits beschrieben, besteht jeder noch so komplexe Körper aus den gleichen Grundelementen, wie sie in der Abbildung 2-2 bezeichnet sind. Diese Bestandteile lassen sich unter dem Begriff "Primitive" zusammenfassen. Die verschiedenen Möglichkeiten, wie mit den Primitiven unter OpenGL gearbeitet werden kann, zeigt die Anlage A.4 "OpenGL Rendermodus".

Rendering

Der Vorgang zur Berechnung eines Bildes wird als Rendering bezeichnet. Wenn der Grafikprozessor diese Berechnung direkt ausführt, dann spricht man vom "Hardware-Rendering". Dabei wird die CPU nicht belastet und es entstehen flüssig ablaufende Animationen. Es kann jedoch auch dazu kommen, dass einige Zeichenbefehle nicht vom Grafikprozessor unterstützt werden. In diesem Fall muss der Hauptprozessor das Ergebnis mittels "Software Rendering" berechnen. Diese Variante ist jedoch deutlich langsamer.

Orientierung in drei Dimensionen

Bei der 3D-Grafik Programmierung kann es grundlegende Vorstellungsprobleme geben. Ein dreidimensionaler Raum, gefüllt mit verschiedensten Objekten, muss auf eine zweidimensionale Fläche projiziert werden. Der Monitor besitzt keine dritte Dimension und dennoch entsteht ein räumlicher Eindruck. Die X- und Y-Achse weisen auf dem Bildschirm, wie gewöhnlich, dessen Breite und Höhe an. Bei OpenGL wurde die negative Z-Achse in die Monitorebene hinein verlegt. Wenn man sich nun den Bildschirm als ein Kameraobjektiv vorstellt, durch welches man in den Raum blickt, so würde man Objekte mit einem positiven Z-Wert nicht sehen. Die Abbildung 2-3 macht deutlich, dass man nur Körper sehen würde, die einen negativen Z-Wert besitzen.

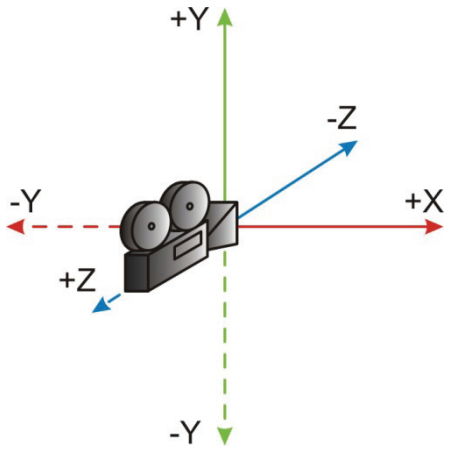


Abbildung 2-3 OpenGL Koordinatensystem

Der Koordinatenursprung befindet sich also genau in der Mitte des Ausgabefensters, in dem der Rendervorgang stattfindet. Dies ist ein Unterschied zur 2D-Grafik Programmierung, wo sich der Koordinatenursprung in der linken oberen Ecke des Fensters befindet. Es ist darüber hinaus hilfreich, wenn man sich eine Kamera vorstellt, die in der virtuellen Welt platziert und bewegt wird. Auch wenn ein Kameraobjekt bei OpenGL nicht wirklich existiert, kann man sich die Bildschirmausgabe in dieser Art vorstellen.

Licht und Material

Die virtuelle Welt so realistisch wie möglich darzustellen, ist eine große Herausforderung. Dabei spielt Licht eine große Rolle. Doch nicht nur Licht bestimmt das Erscheinungsbild. Auch das Material eines Körpers beeinflusst dessen Aussehen. So reflektiert beispielsweise ein Stück Plastik das Licht anders wie eine polierte Metalloberfläche. Diese zwei Bestandteile, Lichtquellen und Materialeigenschaften, können mit OpenGL nachgestellt werden. Prinzipiell ist ein gerendertes Objekt auch ohne Licht sichtbar, sobald es eine andere Farbe wie der Hintergrund besitzt. In diesem Fall fehlt allerdings der räumliche Eindruck, der erst durch eine Lichtquelle erzeugt wird. Die Abbildung 2-4 macht diesen Unterschied deutlich. Die linke Kugel wurde im Gegensatz zu ihrem rechten Ebenbild ohne eine Lichtquelle gerendert.

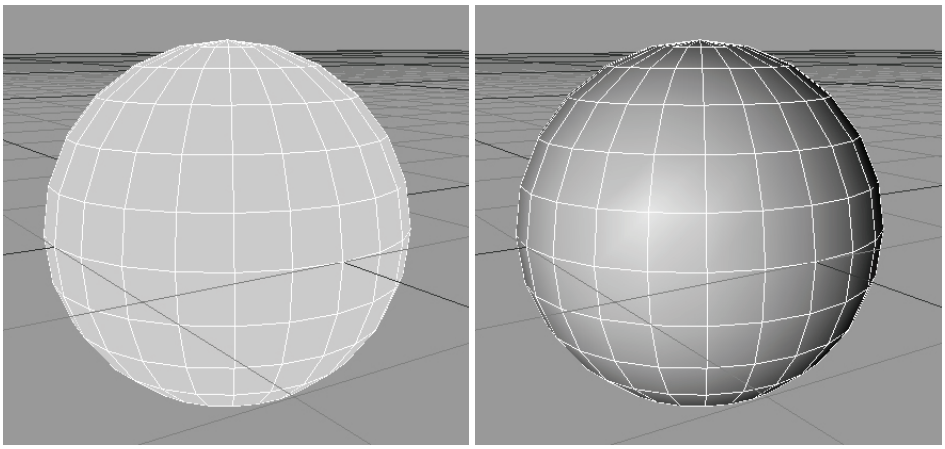


Abbildung 2-4 3D-Szene ohne und mit Lichtquelle

Weiterhin sollte man wissen, dass man standardmäßig acht Lichtquellen zur Verfügung hat, die man konfigurieren und gleichzeitig einsetzen kann. Jede Lichtquelle kann separat konfiguriert werden. Neben der Position gibt es noch weitere wichtige Parameter. Dazu zählen die drei Lichtarten, die jede Lichtquelle simultan aussenden kann. Das „Umgebungslicht“ wird als *Ambient* bezeichnet. In der Realität ist man nahezu überall von diesem Licht umgeben. Bestes Beispiel ist

ein Morgengrauen mit Hochnebel. Die genaue Herkunft des Lichts lässt sich nicht exakt berechnen, daher wird bei OpenGL nur eine konstante Helligkeit und Farbe für diese Art des Lichts bestimmt. Durch das *Ambient* Licht allein entsteht jedoch noch kein räumlicher Eindruck, da es gleichmäßig aus allen Richtungen scheint.

Der zweite Lichtparameter nennt sich *Diffuse* und bezeichnet das direkte Licht, was von der Quelle auf einen Körper fällt. Es sollte den Hauptteil des ausgesendeten Lichtes bilden. Der Einfallswinkel dieses Lichtes auf eine Oberfläche bestimmt dessen Helligkeit. Wenn sich die Fläche von der Lichtquelle abwendet, dann erscheint sie dunkler als eine Fläche, die zu der Lichtquelle senkrecht steht. Der Betrachtungswinkel auf eine Oberfläche und das dem-entsprechende Ergebnis des reflektierten Lichtes, sind auch von der dazu-gehörigen "Normalen" abhängig, die zu dem betreffenden Polygon gehört. Auf Seite 15 wird dieser Begriff ausführlicher beschrieben.

Mit *Specular* wird in der Welt von OpenGL der letzte Beleuchtungsparameter bezeichnet. Hierbei handelt es sich um Glanzlicht. Der Glanzanteil einer Lichtquelle lässt einen Körper aus einer bestimmen Richtung besonders hell erscheinen. Hat dieser Parameter einen hohen Wert, wirken Oberflächen wie poliertes Metall oder Glas. Ist das *Specular* Licht nur sehr gering eingestellt, dann fällt diese Art des Lichtes kaum ins Gewicht.

Für die acht Lichtparameter *GL_LIGHT0* bis *GL_LIGHT7* kann man die Intensität der Lichtarten für jeden Farbkanal getrennt einstellen. Auf diese Weise lassen sich leicht bunt gefärbte Lichtquellen realisieren. Es gibt noch eine Reihe weiterer Einstellmöglichkeiten für OpenGL-Lichtquellen. Dazu zählt die Abnahme der gesamten Lichtintensität mit der Entfernung. Diese kann deaktiviert werden, linear oder quadratisch abnehmen.

Weiterhin kann man zwischen einer kugelförmigen Lichtquelle und einem Lichtkegel, ähnlich einer Taschenlampe, unterscheiden. Die drei Parameter, die in Abbildung 2-5 zu sehen sind, müssen definiert werden, wenn die Lichtquelle als "Spot" eingesetzt werden soll.

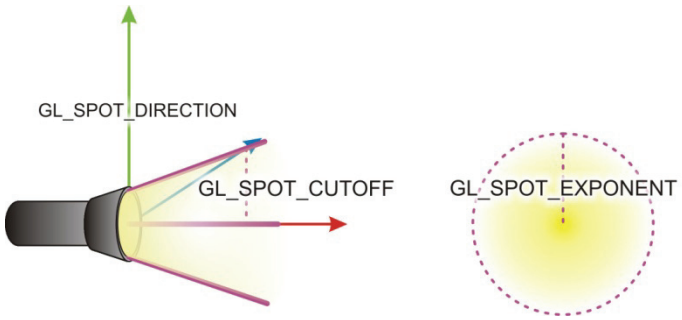


Abbildung 2-5 Parameter eines Lichtkegels

GL_SPOT_DIRECTION ist ein Vektor, angegeben in den Weltkoordinaten, der die Richtung des Lichtkegels festlegt. Um den Öffnungswinkel des Lichtkegels einzustellen, muss man *GL_SPOT_CUTOFF* definieren. Aufgrund der Tatsache, dass sich dieser Wert nur auf den halben Lichtkegel bezieht, ist zu beachten, dass es sich bei einem maximalen Winkel von 180 Grad, wieder um eine kugelförmige Lichtquelle handelt. Die Abschwächung des Lichtkreises, der auf der Oberfläche eines Körpers zu sehen ist, wird mit *GL_SPOT_EXPONENT* definiert. Standard-mäßig ist dieser Wert auf Null gesetzt, woraus eine gleichmäßige Ausleuchtung des gesamten Bereichs resultiert, der in der Abbildung 2-5 auf der rechten Seite gekennzeichnet ist.

Normalen

Ein wichtiger Punkt bei der Beleuchtung von Oberflächen sind deren Normalen. Dieser Begriff stammt aus der Vektorrechnung und sollte daher bekannt sein. Eine Normale ist unter OpenGL ein nicht sichtbarer Richtungsvektor, der die Außenseite eines Polygons festlegt und die genaue Richtung, in welche die

Lichtstrahlen reflektiert werden. Die Normale sollte stets senkrecht zur Oberfläche stehen. Es besteht die Möglichkeit eine Normale für jeden Vertex einzeln, für jedes Polygon oder als Durchschnittswert für mehrere Polygone zu berechnen. Je nach dem, wie viele Normalen auf der gesamten Oberfläche verteilt werden, können dieses abgerundet werden. Darüber hinaus sollte die Normale stets auf eins normiert werden, da die Länge die Intensität der Lichtreflexion angibt. In der Abbildung 2-6 wurden die Normalen des Körpers als gelbe Linien sichtbar dargestellt. Wie man sieht, wurde für jedes Polygon eine Normale berechnet.

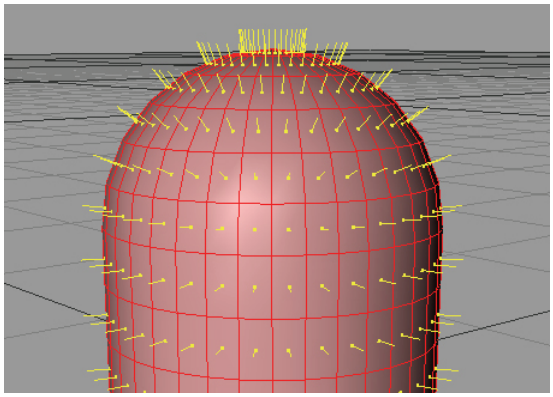


Abbildung 2-6 Normalen eines 3D-Objektes

Texturen

Unter einer Textur versteht man eine ganz normale Bitmap, die auf die Oberfläche eines 3D-Körpers aufgebracht wird, damit ein fotorealistischer Eindruck entsteht. Auf diese Weise lassen sich mit wenigen Polygonen und einer Textur detaillierte Objekte erstellen. Die Abbildung 2-7 zeigt einen Würfel, der mit einer Holztextur versehen wurde. Durch diese wirkt der sehr einfach modellierte Körper aufwändig gestaltet. Das blaue Drahtgittermodell des Körpers wurde zur Veranschaulichung der vorhandenen Polygone mit gerendert.

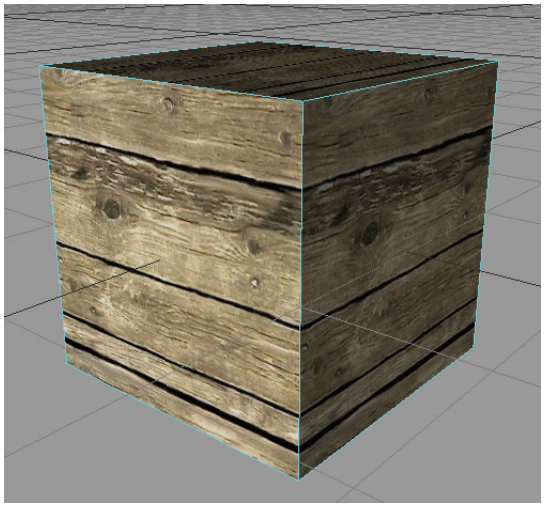


Abbildung 2-7 Holztextur auf einem 3D-Würfel

Die Textur darf allerdings nicht mit dem Begriff Material verwechselt werden. Die Materialeigenschaften eines Objektes stehen nur mit der Beleuchtung im Zusammenhang, wogegen eine Textur auf das Material keinen Einfluss hat.

OpenGL Matrizen

In der Welt von OpenGL gibt es drei wichtige Matrizen, die für das resultierende Monitorbild mit verantwortlich sind. Diese 4x4 Matrizen bestehen aus Zahlenwerten, denen weitere Werte hinzugefügt oder entnommen werden können. Mit Hilfe der so genannten Transformationsmatrizen, werden dann alle Körper und Objekte der virtuellen Umgebung beeinflusst.

Die wichtigste ist die so genannte *ModelView* Matrix. Diese verschiebt, dreht oder skaliert jeden Vertex, also jeden Eckpunkt, aus dem ein Körper besteht. Diese Matrix hat während des Zeichenvorgangs auf alle Objekte den gleichen Effekt, bis an dem Matrixstapel etwas verändert wird. Dieses Zustandsverhalten kann man sich unter anderem beim Platzieren von Objekten leicht zunutze machen. Auf die Arbeitsweise mit dieser Matrix wird im Kapitel 4.3 "Anwendung der Visualisierungsbibliothek" näher eingegangen.

Die zweite Matrix ist die *Perspective* Matrix. Sie bestimmt, wie der Name schon sagt, die Perspektive, in der die 3D-Szene gerendert wird. Damit kann man unter anderem die Brennweite der virtuellen Kamera festlegen. Die letzte Matrix dient ausschließlich der Manipulation von Texturen. Somit lässt sich eine Textur auf verschiedenen Oberflächen unterschiedlich darstellen, ohne eine zweite Textur hinzu laden zu müssen. Ihre Funktionsweise gleicht dabei der *ModelView* Matrix. Sowohl die *Perspective*, als auch die *Texture* Matrix sollen an dieser Stelle nur am Rande erwähnt werden, da sie für das Verstehen der 3D-Grafik Programmierung nicht weiter erforderlich sind.

SceneGraph

Allgemein bezeichnet der Begriff "SceneGraph" eine objektorientierte Datenstruktur, die in der Lage ist grafische Inhalte darzustellen und zu verwalten. Einen konkreten Funktionsumfang, den ein SceneGraph liefern muss gibt es nicht. [SG 09] Der Begriff soll in dieser Arbeit als ein konkretes Steuerelement verstanden werden, welches in der Entwicklungsumgebung *Visual Studio* eingesetzt werden kann. Mit Hilfe dieser Instanz soll die Entwicklung dreidimensionaler Applikation auf Basis des .Net Framework vereinfacht und komfortabler gestaltet werden.

3 Die grafische Oberfläche der Simulationssoftware

3.1 Konzept

Die Abbildung 3-1[8] zeigt das grobe Fachkonzept der Simulationsanwendung, welches in Toni Dirk Großmanns Diplomarbeit erläutert wurde. Wie man sieht, ist bereits in dieser Konzeptionsphase die "Anzeige" (rot hervorgehoben) als eigenständige Komponente vorgesehen. Dieser Bestandteil soll alle Ausgaben, beispielsweise Simulationsvorgänge und zusätzliche Informationen, darstellen. Die beiden Strukturen "Simulation" und "Simulationsszenario" interagieren direkt über definierte Schnittstellen mit der "Anzeige". Die darstellende Schicht des Simulators/Demonstrators soll nun in dieser Arbeit detailliert erarbeitet und realisiert werden.

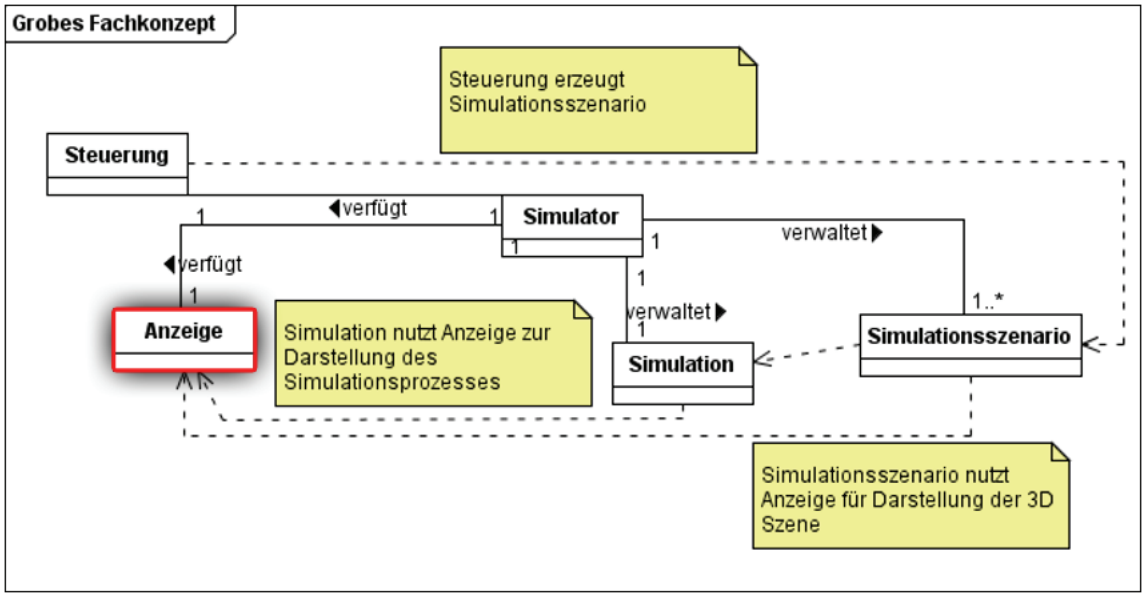


Abbildung 3-1 Fachkonzept Simulator/Demonstrator

[8] Quelle: [Großmann 09] S. 41

3.2 Anforderungsanalyse

3.2.1 Zweck und Umfang

Der Simulator/Demonstrator erfüllt, gemäß seines Namens, mehrere Aufgaben. Zum einen soll er Vorgänge in Netzwerken simulieren und darstellen können. Zum anderen soll er in Zukunft auch die Fähigkeit besitzen, reale Netzwerke abbilden zu können. Es ist nicht ausgeschlossen, dass weitere Funktionen hinzukommen. Die Fähigkeit, auf ein reales Sensornetz Einfluss zu nehmen, ist daher nahe liegend. Damit ist gemeint, dass es Funktionen zum Messen und Aufzeichnen diverser Parameter eines Netzwerkknotens geben könnte. Die möglichen Einsatzgebiete der Software sind also sehr vielseitig. Eingesetzt als reine Simulationssoftware, bis hin zur Überwachungssoftware, die auf einer Basisstation installiert sein könnte, ist alles denkbar. Somit wird klar, dass sich der Umfang während der Entwicklungsphase stetig weiterentwickelt und die darstellende Schicht mit dem Ausbau der Funktionen Schritt halten muss.

3.2.2 Funktionale Anforderung

In der Anlage A.1 "Pflichtenheft der Softwareoberfläche" werden die Produktfunktionen einzeln aufgelistet. Die wichtigsten darin aufgeführten Merkmale werden nun erläutert. Die Software soll die Fähigkeit besitzen, dass alle Inhalte in beliebiger Anordnung angezeigt, platziert, verschoben oder verborgen werden können. Damit die Oberfläche auch in ferner Zukunft leicht erweiterbar ist, sollen alle Fensterinhalte unabhängig von einander dargestellt werden. Es ist also erforderlich, für jeden Aufgabenbereich ein separates Fenster anzulegen. Künftige Inhalte können dann, dem zu Grunde liegendem Konzept entsprechend, leicht implementiert werden. Diese Vorgehensweise kommt der Anforderung entgegen, dass der Anwender diese Inhalte frei platzieren können muss. Die getroffene

Einstellung soll sich darüber hinaus speichern lassen und bei Bedarf soll eine andere geladen werden können.

Die Darstellung des Simulationsprozesses soll zwei- und dreidimensional möglich sein. Auf älteren PC-Systemen kann es zu Schwierigkeiten bei der 3D-Darstellung kommen, da diese nicht über die notwendige Leistungsfähigkeit verfügen, um die Bilder in ausreichender Geschwindigkeit zu berechnen. Um dennoch eine lauffähige Software zu gewährleisten, muss die Möglichkeit bestehen, dass die 3D-Komponente eine vereinfachte Visualisierung ausführt, oder komplett deaktiviert werden kann.

Es müssen weitere Diagramme zur Analyse und Auswertung bereit gestellt werden. Dazu gehören eine detaillierte Aufschlüsselung der Batteriewerte jedes Knotens, die Überlebensrate des Netzwerkes und weitere Tabellen, die Informationen über die Netzwerkknoten anzeigen. Hinzu kommt, dass der Nutzer mit möglichst wenigen Interaktionen die wichtigsten Programmabläufe starten können muss. Es ist demnach darauf zu achten, dass diese wichtigen Funktionen, mit bestenfalls einem Mausklick, erreichbar sind.

Ein weiterer Punkt ist die Mehrsprachigkeit, denn darauf kann im Zuge der Globalisierung nicht mehr verzichtet werden. Die Sprache, in der das Programm erscheint, soll während der Laufzeit umgeschaltet werden können. In der Anlage A.1 befinden sich alle detaillierten Produktfunktionen die für das Pflichtenheft der Softwareoberfläche erarbeitet wurden.

3.2.3 Nicht funktionale Anforderung

Der Anwender erwartet, dass gewisse visuelle Quasistandards eingehalten werden. Diese beziehen sich unter anderem auf die Menüführung oder das Erscheinungsbild von Dialogen und Einstellungsfenstern. Der Hauptcontainer der

Oberfläche sollte individuell anpassbar sein, was schon aus den funktionalen Anforderungen hervor geht. Damit wird der Komfort des Programms gesteigert, da der Nutzer sämtliche Inhalte nach seinen Vorlieben oder für einen speziellen Zweck ausrichten kann. Eine daraus resultierende hohe Nutzerakzeptanz soll eine lange Lebensdauer der Anwendung gewährleisten. Die Oberfläche soll darüber hinaus optisch modernen Ansprüchen genügen. Dies betrifft eine Reihe grafischer Komponenten. Für zahlreiche Steuerelemente müssen beispielsweise qualitativ hochwertige und selbst erklärende Icons gestaltet werden.

3.2.4 Anforderungen an Qualität und Performanz

Die Absturzsicherheit zählt zu den wichtigsten Qualitätsmerkmalen jeder Computersoftware. Während der Laufzeit des Simulators/Demonstrators darf es zu keinem kritischen Absturz kommen. Es muss allerdings unterschieden werden, ob ein Fehler direkt von der GUI verursacht wird, oder durch den Programmkern hervorgerufen wird. Bei jedem Fehler muss die Oberfläche mit einer entsprechenden Meldung reagieren, oder intelligent umgeleitet werden. Das bedeutet, dass Teile der Oberfläche gesperrt werden müssen, falls diese zu einem bestimmten Zeitpunkt der Laufzeit einen Fehler verursachen würden.

Die Geschwindigkeit, mit welcher bestimmte Prozesse ablaufen, ist ebenfalls ein Zeichen für die Qualität von Programmen. Da die Konzeption des Simulators/ Demonstrators eine saubere Trennung von Berechnungen und Visualisierung vorsieht, hängt die Performanz stark von den Algorithmen ab, die unterhalb der Darstellungsschicht ablaufen. Für die Oberflächenprogrammierung ist es also von Bedeutung, dass es bei der Berechnung von Grafiken nicht zu weiteren Verzögerungen kommt. Die zweidimensionale Darstellung des Simulationsprozesses ist für heutige PC-Systeme keine leistungskritische Aufgabe und dürfte die Gesamtgeschwindigkeit daher nicht weiter beeinträchtigen. Im Gegensatz dazu

ist die 3D-Visualisierung, auch für moderne Computer, eine anspruchsvolle Aufgabe. Hier muss eine Möglichkeit gegeben sein, die 3D-Anzeige zu vereinfachen oder einzustellen.

3.2.5 Besonderheiten für Entwickler

Im Gegensatz zu den Anforderungen die ein Nutzer an die Software stellt, ergeben sich im Hinblick auf künftige Weiterentwicklungen ganz andere Schwerpunkte. Der Oberfläche muss ein Konzept zu Grunde liegen, sodass neue Inhalte problemlos und ohne großen Aufwand aufgenommen werden können. Die Kommunikation zwischen Programmoberfläche und Kern der Anwendung muss also über sauber definierte Schnittstellen erfolgen. Wenn der modulare Aufbau der Oberfläche konsequent eingehalten wird, dann verbessert sich somit auch die Übersichtlichkeit des gesamten Softwareprojektes.

Die Mehrsprachigkeit der Anwendung soll vollständig vom Projekt abgekoppelt werden. Damit entfällt, sowohl für den Programmierer, der Aufwand des Übersetzens, als auch für den Übersetzer, das Erlernen von Entwicklungs-umgebung und Programmierkenntnissen.

3.3 Design der Oberfläche

3.3.1 Voraussetzungen

Für das Konzept, die Oberfläche modular zu strukturieren, gibt es zahlreiche Lösungsvarianten. Allgemein ist die Anwendung bei allen diskutierten Fällen eine MDI-Anwendung[9]. Dabei existiert ein Hauptfenster, welches als „Parent-Form“ bezeichnet wird. Dieses kann nun beliebig viele weitere Fenster, so genannte „Child-Forms“, in sich aufnehmen. Die untergeordneten Fenster stellen dann die

[9] Multiple Document Interface - Anwendung

eigentlichen Programmabläufe dar. Es betrifft nur eine einzige Einstellung, die ein normales Programmfenster zu einem MDI-Container werden lässt. Die Funktionalität, dass man diese Kind-Fenster einsetzen und verwalten kann, ist bei dem .Net Framework nur sehr beschränkt vorhanden. Standardmäßig bietet es die Möglichkeit, dass man sämtliche Child-Forms hinter einander anzuordnen und über Registerkarten jeweils ein Fenster in den Vordergrund stellen kann. Die zweite mögliche Ansicht wäre, alle Fenster lose vor den Hauptcontainer zu legen. Dabei geht jedoch jeder Komfort verloren, da sich die Fenster teilweise überdecken können und es keinerlei feste Anhaltspunkte gibt. Daraus folgt die Konsequenz, dass ein Funktionspaket entworfen werden muss, dass den MDI-Container in seinen Fähigkeiten deutlich verbessert.

Ein überaus nahe liegendes Vorbild ist die Entwicklungsumgebung *Visual Studio* von Microsoft. Das hier verwendete MDI-Konzept wird im Allgemeinen als „DockContent“ bezeichnet. Die Oberfläche dieser Software entspricht modernen Anforderungen, ist leicht verständlich und lässt sich problemlos durch den Anwender anpassen. Es erweitert die Eigenschaften der Child-Forms wie folgt.

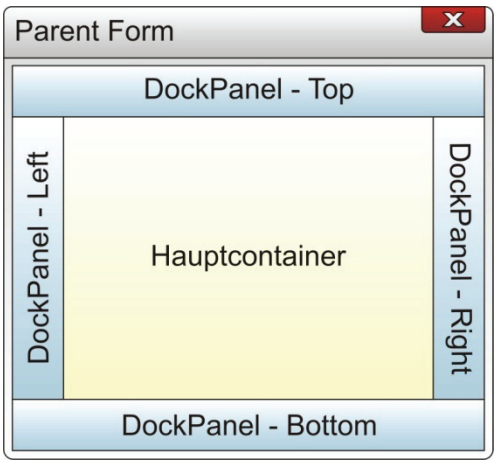


Abbildung 3-2 DockContent Schema

Die Abbildung 3-2 zeigt die möglichen Zustände, die praktisch jedes Kind-Fenster annehmen kann. Die Fenster, die sich im Hauptcontainer befinden, lassen sich über Registerkarten sortieren und zusätzlich kacheln. Die Fenster, die einem "Panel" am Rand zugeordnet werden, lassen sich dort fixieren und zusätzlich ein- beziehungsweise ausblenden. Die Funktion des Ein- und Ausblendens der Fenster an den Rändern hat noch einen besonderen Vorteil. Dadurch kann eine Menge Platz eingespart werden, da beim Ausblenden nur der Titel des Fensters stehen bleibt. Fährt man mit der Maus darüber, bewegt sich der Fensterinhalt vom Rand wieder herein und kann genutzt werden. Alle integrierten Fenster können direkt mit der Maus verschoben werden. Es gibt bei jeder Bewegung einer Form eine visuelle Vorschau, die anzeigt, wo das Fenster voraussichtlich neu verankert werden wird.

Es spielt keine Rolle, wie viele Fenster angezeigt, oder wo diese platziert werden sollen. Alle Inhalte liegen in einer klaren Anordnung vor. Es wurde bereits während der Anforderungsanalyse klar, dass diese überaus nützlichen Funktionen, die von *Visual Studio* vorgeführt werden, wegweisend für den Simulator/ Demonstrator sein müssen.

3.3.2 Lösungsmöglichkeiten

Eigenentwicklung

Die erste Option ist es nun, sämtliche Funktionalitäten in Eigenleistung zu entwickeln. Um die Übersichtlichkeit des gesamten Projektes zu wahren, muss das Funktionspaket in jedem Fall in eine Klassenbibliothek ausgelagert werden. Sämtliche Positionierungen der Fenster und das Andockverhalten müssen berücksichtigt werden. Es gibt eine Vielzahl von Ereignissen, die unter anderem vom Benutzer ausgelöst werden. All diese müssen behandelt und entstehende Fehler abgefangen werden. Viele der visuellen Ereignisse und Animationen

würden nicht absehbare Programmierzeit beanspruchen. In Effekte, zum Beispiel die Bewegungen bei dem Ein- und Ausblenden, müsste demnach ein großer Aufwand investiert werden, damit diese fehlerfrei ablaufen. Es muss darüber hinaus ein eigenes XML-basiertes Konzept zum Speichern und Laden der Einstellung entworfen werden. Wie man sieht, stellt sich dieser Lösungsansatz als enorm aufwändig heraus.

Die Zeit, die man für den Entwurf benötigt, um ein qualitativ ansprechendes Ergebnis zu erhalten, würde bereits eine eigenständige Abschlussarbeit rechtfertigen. Die Aufgabenstellung sieht jedoch noch eine Vielzahl weiterer Tätigkeiten vor, die bearbeitet werden müssen. Daher wird an dieser Stelle nach einer zeitlich effizienteren Lösung gesucht. Nach einigen Recherchen im Internet fanden sich bereits zahlreiche fertige Konzepte. Diese sollte man allerdings nur dann in Betracht ziehen, wenn die Quellen einsehbar sind und die Lizenzen eine freie Verwendung gewährleisten. In den folgenden Abschnitten werden drei weitere frei zugängliche Klassenbibliotheken diskutiert.

AvalonDock

Die erste Variante nennt sich „AvalonDock.dll“ [Avalon 09] und basiert vollständig auf der Windows Presentation Foundation. WPF ist ein reines Grafik-Framework, welches innerhalb des .Net Frameworks ab Version 3.0 unterstützt wird. Die dabei eingesetzte Programmiersprache ist allerdings XAML. Sie dient allein der Deklaration der hierarchischen Anordnung aller Steuerelemente. Damit trennt das WPF Framework zwangsläufig die darstellende Schicht von der Programm-logik. WPF bietet darüber hinaus eine Vielzahl interessanter grafischer Möglichkeiten, da bei der Darstellung aller Steuerelemente, die Hardware-unterstützung durch die GPU genutzt wird. Zahlreiche Effekte, wie Schattierung-

en, oder Dienste zum Darstellen von Vektorgrafiken oder Transformationen von Bildern werden angeboten.

Ein eindeutiger Nachteil ist, dass man sich zunächst in die Syntax von XAML einarbeiten muss. Der Kern des Simulator/Demonstrator, der nach wie vor mit C# entwickelt wird, soll schließlich sauber mit der Oberfläche verknüpft werden. Schon beim Start des mitgelieferten Beispiels, welches die AvalonDock Klassenbibliothek verwendet, wurden eine Reihe weiterer Nachteile deutlich. Zum einen können die Child-Forms keine eigenen Titelsymbole aufnehmen, was die gesamte Applikation bei steigender Zahl der Fenster schnell unübersichtlich werden lässt. Davon abgesehen ist die DLL nicht fehlerfrei, und da es sich hierbei um keine kostenpflichtige Lizenz handelt, entfällt auch jede Gewährleistung.

Die Abbildung 3-3 zeigt eine Momentaufnahme während das rechts fixierte Fenster „TextBox1“ zum rechten Rand hinaus fährt. Während dieser Bewegung wird der Rest der Anwendung nicht neu gerendert. Das Fenster bleibt in seiner ursprünglichen Breite stehen und wird während der Animation immer wieder von sich selbst überdeckt. Von der Verwendung dieser fehlerhaften Klassenbibliothek muss selbstverständlich abgesehen werden. Somit kommt diese DLL als Lösungsmöglichkeit nicht in Frage.

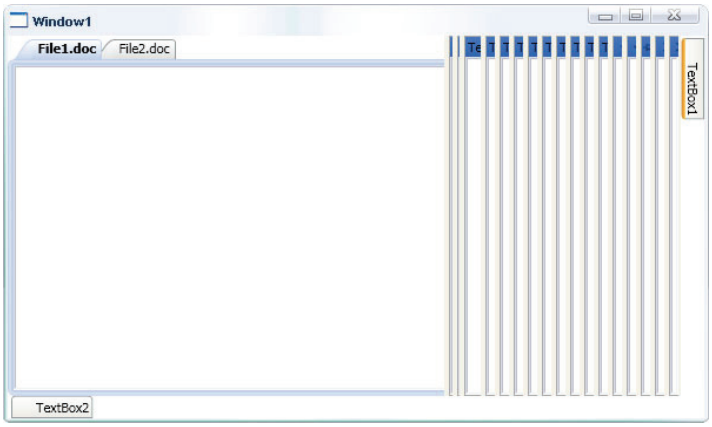


Abbildung 3-3 Testapplikation "AvalonDock"

DockingControls

Zum Vergleich wurde weiterhin die „DockingControls.dll“ [DC 09] getestet. Diese DLL setzt auf die Verwendung des .Net Frameworks und die darin enthaltenen klassischen Steuerelemente. Die Klassenbibliothek stellt dafür ein Steuerelement „DockingManagerControl“ zur Verfügung. Bei der Entwicklung einer Demoanwendung viel auf, dass es sich nicht um ein konsequent umgesetztes MDI-Konzept handelt, denn alle im Zentrum platzierten Inhalte können vom Nutzer nicht mehr bewegt werden. In der Abbildung 3-4, die das Testprogramm zeigt, ist dieser Bereich rot markiert. Es ist bei diesem frei zugänglichen Projekt vorgesehen, im Zentrum nur fixierte Inhalte zu platzieren. An allen Rändern können dann die untergeordneten Fenster implementiert werden, die sich andocken und bewegen lassen. Alle Child-Forms, die für den Nutzer manipulierbar sein sollen, müssen lediglich beim DockingManagerControl an einem der vier DockPanels angemeldet werden, um neben dem Hauptcontainer zu erscheinen.

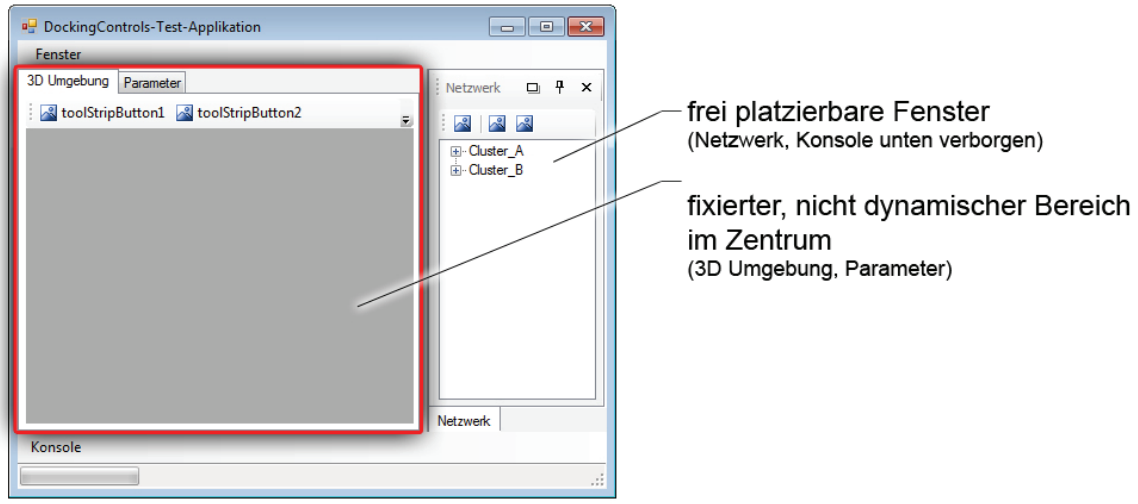


Abbildung 3-4 Testappliaktion "DockingControls"

Die Handhabung der Oberfläche stellte sich im Gegensatz zu ihrer einfachen Programmierung als ungewohnt und unvollständig heraus. Es fehlten beispielsweise einige wichtige Funkionen und die gesamte Oberfläche flimmert leicht, wenn man an der Fensterposition eines der DockPanels etwas verändert. Zusammenfassend kann man also über diese Klassenbibliothek sagen, dass sie ebenfalls unzureichend ist, da die Gebrauchstauglichkeit nicht den Anforderungen des Simulators/Demonstrators entspricht.

Weifen Luo

Eine viel versprechende Lösung war die Klassenbibliothek von Weifen Luo. Diese wurde bereits in zahlreichen Internet-Foren empfohlen und als fehlerfrei bewertet. In dieser Bibliothek werden alle Kind-Fenster mit *DockContent* bezeichnet. Die „WeifenLuo.WinFormsUI.Docking.dll“ [Luo 09] liefert dazu ebenfalls ein Steuerelement mit, welches hier den Namen *DockPanel* trägt. Die DLL von Weifen Luo steht unter der MIT Lizenz[10] im Internet zur freien Verfügung und kann daher uneingeschränkt genutzt, verbreitet oder modifiziert werden. Die als Textdatei mit gelieferte Originallizenz befindet sich in den Anlagen dieser Arbeit unter A.2 "Die MIT Lizenz".

Die Implementierung der Funktionen war nur unwesentlich aufwändiger, als bei den anderen analysierten Klassenbibliotheken. Alle Fenster, welche die Fähigkeit der Verankerung besitzen sollen, müssen ihre neuen Funkionen von einer Basisklasse aus der DLL erben. Danach können die Forms, die als *DockContent* deklariert worden sind, normal instanziiert werden. Sie benötigen bei ihrem Aufruf lediglich die Zusatzinformation, welchem *DockPanel* Objekt sie angehören sollen.

[10] diese Lizenz stammt aus dem Massachusetts Institute of Technology

Der Anwender kann nun während der Laufzeit, durch eine einfache Mausbewegung oder einen Doppelklick auf den Titel der Form, jedes Fenster aus seiner Verankerung lösen und an einer beliebigen Stelle wieder neu andocken. Die Animation der *DockContent* Elemente, die am Rand ein- und ausgeblendet werden, funktioniert reibungslos. Die Möglichkeit, die Einstellung der Fenster zu speichern und wieder zu laden, wird außerdem bereits mitgeliefert. Die Hierarchien und Abhängigkeiten der einzelnen *DockContent* Fenster werden mittels XML-Syntax persistent gespeichert.

Standardmäßig verliert jeder *DockContent* sämtliche Daten beim Betätigen des Schließen-Buttons und kann nicht wieder hergestellt werden. Dies ist natürlich unbedingt zu vermeiden. Um die Child-Forms wieder herstellen zu können genügt es, alle *DockContent* Objekte in eine Liste einzutragen. Diese Liste kann an einen Menüpunkt in der Hauptform gekoppelt werden. Wird dort ein *DockContent* Fenster ausgewählt, so erscheint es wieder an seiner alten Position. Die darin enthaltenen Daten gehen nicht verloren, da die Möglichkeit besteht, den *DockContent* bei einem Klick auf das Schließen-Kreuz nur zu verbergen, anstatt zu beenden. Diese Funktionalität muss allerdings in Eigenleistung hinzugefügt werden.

Der Vergleich aller Lösungsansätze hat gezeigt, dass die DLL von Weifen Luo am besten geeignet ist. Bei der Programmierung sind keine Komplikationen aufgetreten. Die Handhabung ist selbsterklärend, fehlerfrei und mit dem Vorbild *Visual Studio* identisch. Es hat sich gezeigt, dass die Bibliothek den vollen geforderten Funktionsumfang liefert, wie er in den Voraussetzungen erarbeitet wurde. Diese Variante besitzt also keine der Schwächen, die bei den anderen Klassenbibliotheken aufgetreten sind. Im folgenden Abschnitt wird genauer erklärt, wie die DLL verwendet wird.

3.3.3 Implementierung der GUI

Die ausgewählte Klassenbibliothek wurde während der Bearbeitungszeit in den Simulator/Demonstrator implementiert. Die nachfolgend vorgeführten Programmcode-Ausschnitte veranschaulichen die genaue Funktionsweise innerhalb der Forschungssoftware. Damit soll nachgewiesen werden, dass die DLL den Anforderungen des Forschungsprojektes entspricht. Da die Quellen der Bibliothek zur Verfügung stehen, ist es durchaus aufschlussreich, diese einmal zu begutachten. Dabei wird der Umfang der WeifenLuo.WinFormsUI.Docking.dll erst deutlich. Sie besteht aus 14 Enumerationen, fünf Interfaces und nicht zuletzt aus 52 Klassen. Hier ist auch das Steuerelement *DockPanel* zu sehen, welches später alle Fenster in sich aufnimmt und verwaltet. Das genaue Zusammenwirken der Strukturen spielt für das Verstehen der Beispielcodes keine weitere Rolle.

Der Projektmappe des Simulators/Demonstrators muss ein Verweis auf die Klassenbibliothek hinzugefügt werden, damit auf deren Bestandteile zugegriffen werden kann. Die IDE von *Visual Studio* erkennt bereits automatisch, dass ein neues Steuerelement mitgeliefert wird und fügt dieses in die Liste der „Toolbox“ ein. Die Anlage A.3 zeigt die Entwicklungsumgebung, in der eine Instanz von *DockPanel* soeben auf die *Form1*, das Hauptfenster der Anwendung, gezogen wurde. Damit steht das Objekt *dockPanel1* bei der Programmierung zur Verfügung.

Im Anschluss müssen weitere Kind-Fenster erstellt werden, die später innerhalb der *dockPanel1* Instanz angezeigt werden sollen. Dabei ist es notwendig, dass all diese Fenster von der *DockContent* Klasse erben. Der zugehörige Namensraum aus der DLL darf dabei nicht vergessen werden. Wenn die Eigenschaft *HideOnClose* auf *true* eingestellt wird, verhindert dies ein Löschen des Fensters,

wenn man auf das Schließen-Kreuz der Child-Form klickt. Der beschriebene Quellcode 3-1 sieht demnach folgendermaßen aus.

```
// der eingebundene Namensraum der DLL
using WeifenLuo.WinFormsUI.Docking;

namespace MyFirstDockPanelExample
{
    // die Kind-Form erbt von DockContent
    public partial class DockingWindow : DockContent
    {
        this.HideOnClose = true;
        // weiterer Code, den das Kind Fenster ausführen soll...
    }
}
```

Quellcode 3-1 Einsatz der DockContent Klasse

In der *Form1* müssen alle neuen Fenster nur noch instanziiert werden. Durch die Vererbung von *DockContent* wurde die *Show()* Methode der *DockingWindow* Klasse erweitert. Dadurch kann die Methode das *dockPanel1* Objekt und weitere Parameter aufnehmen. Wird nun die *Show()* Methode ausgeführt, dann erscheint die Instanz innerhalb der *dockPanel1* Oberfläche an der angegeben Position.

Die Child-Forms lassen sich in zwölf verschiedenen *DockState* Einstellungen initiieren. Hinzu kommt, dass sich die Kind-Fenster auch als eigenständige Fenster mit einer definierten Größe und Position öffnen lassen. Zusätzlich können sie auch verschachtelt werden, was die große Flexibilität der Bibliothek deutlich macht. Der Quellcode 3-2 beschreibt, wie ein Hauptfenster mit vier Child-Forms an unterschiedlichen Positionen geöffnet wird. Die grünen Kommentare beschreiben wo diese liegen.

```
using WeifenLuo.WinFormsUI.Docking;

namespace MyFirstDockPanelExample
{
    public partial class Form1 : Form
    {
        // 4 Kind-Fenster instanzieren
        DockingWindow dockingWindow1 = new DockingWindow();
        DockingWindow dockingWindow2 = new DockingWindow();
        DockingWindow dockingWindow3 = new DockingWindow();
        DockingWindow dockingWindow4 = new DockingWindow();
```

```
        // Konstruktor des Hauptfensters
        public Form1() {}

        private void Form1_Load(object sender, EventArgs e)
        { // Hauptfenster wird geladen

            // das Fenster 1 erscheint als zentrales Dokument
            dockingWindow1.Show(dockPanel, DockState.Document);

            // das Fenster 2 erscheint rechts angedockt
            dockingWindow2.Show(dockPanel, DockState.DockRight);

            // das Fenster 3 erscheint unten, verborgen und angedockt
            dockingWindow3.Show(dockPanel, DockState.DockBottomAutoHide);

            // das Fenster 4 erscheint als losgelöstes Fenster
            // Position und Abmessungen entsprechen dem übergebenen Rechteck
            dockingWindow4.Show(dockPanel, new Rectangle(10, 10, 450, 300));
        }
    }
}
```

Quellcode 3-2 Möglichkeiten der *Show()-Methode*

Für die Simulationssoftware mussten alle integrierten Fenster in ein Listenobjekt eingetragen werden. Die Listeneinträge wurden an den Hauptmenüpunkt „Fenster“ mit einer *foreach* Schleife übergeben, damit verborgene Fenster, bei einem Klick auf den Menüpunkt, wieder sichtbar gemacht werden können.

Die Abbildung 3-5 zeigt eine Rohfassung der Softwareoberfläche, wo momentan die Child-Form „3D Umgebung“ unter dem „Parameter“ Fenster neu angedockt wird. Sobald man das Fenster mit der Maus bewegt, werden diverse Ankersymbole sichtbar, die anzeigen , wo das Fenster "3D Umgebung" angedockt werden kann. Zusätzlich sieht man eine transparente Fläche die eine genaue Vorschau gibt, wo sich das Kind-Fenster beim Loslassen der Maus befinden wird.

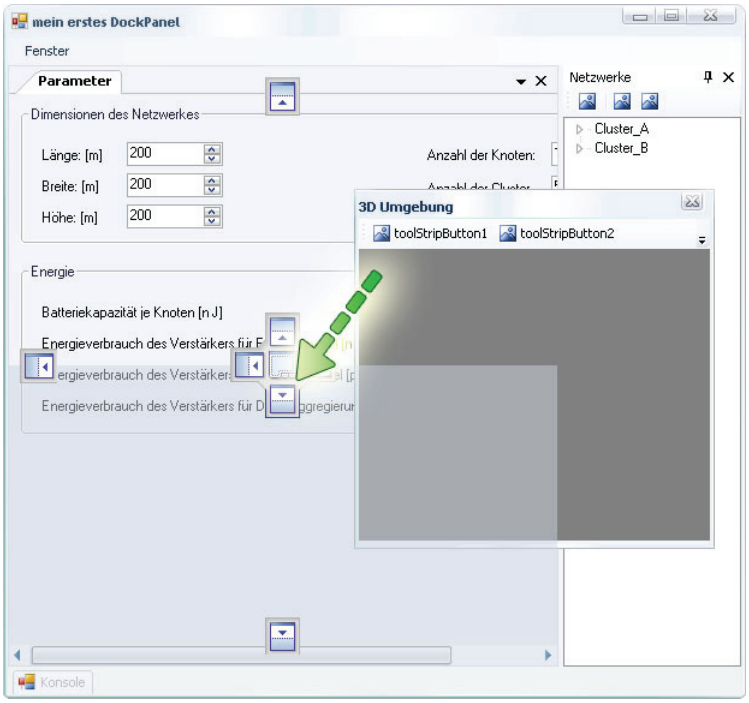


Abbildung 3-5 Testapplikation "DockPanel"

Auch der Automatismus zum Speichern und Laden ist in wenigen Schritten implementiert. Man benötigt eine Zeichenkette, die den absoluten Pfad zu einer XML-Datei bereit stellt, in der die Oberflächeninformationen gehalten werden sollen. Das *dockPanel1* Objekt besitzt eine *LoadFromXml()* und eine *SaveAsXml()* Methode. Zum Speichern genügt es, den Speicherpfad inklusive Dateinamen zu übergeben. Beim Laden der Einstellung wird es ein wenig komplexer. Man benötigt zusätzlich ein Hilfsobjekt aus der DLL. Diesem muss eine selbst erstellte Methode übergeben werden, die alle *DockContent* Objekte mit den Einträgen aus der XML Datei vergleicht. Entspricht das Fenster einem Eintrag in der XML Baumstruktur, so wird dieses, der gespeicherten Einstellung entsprechend, geöffnet.

3.3.4 Fazit zur Softwareoberfläche

Die Problematik, die Anordnung der Fenster in Echtzeit verändern zu können, wurde schließlich mit einer externen DLL gelöst. Mit Hilfe der Klassenbibliothek

von Weifen Luo ist es möglich, die Funktionalitäten des MDI-Containers enorm zu verbessern. Die DLL steht dem Forschungsprojekt uneingeschränkt zu Verfügung und arbeitet fehlerfrei. Darüber hinaus ist die Implementierung leicht nachvollziehbar.

Die Steuerung richtet sich nach modernen Quasistandards, was maßgeblich zur Akzeptanz der der Simulationsanwendung beiträgt. Die dem Anwender damit gegebenen Möglichkeiten, erfüllen die Anforderungen an die Gebrauchstauglichkeit und Angemessenheit an den Umfang der Software. Zudem ist die Anwendung in hohem Maße individualisierbar. Die heraus gearbeiteten Produktfunktionen der Anlage A.1 sind bei der Entwicklung der Softwareoberfläche umgesetzt worden. Die Abbildung 3-6 zeigt die wichtigsten realisierten Inhalte, die für die Darstellung der Simulation von drahtlosen Netzwerken von Bedeutung sind.

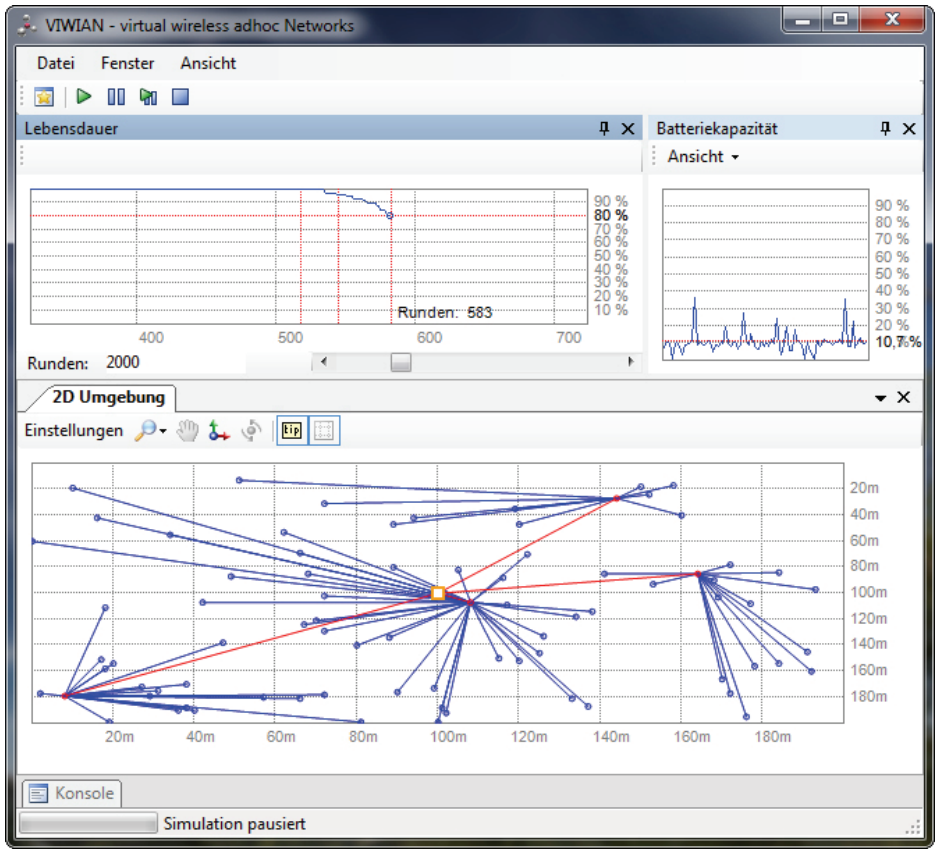


Abbildung 3-6 Momentaufnahme der Simulationssoftware

Für jeden Aufgabenbereich der Simulationsanwendung, zum Beispiel die "2D Umgebung" oder die Diagramme "Lebensdauer" und "Batteriekapazität", sind eigenständige Fenster realisiert worden. Eine genauere Beschreibung, sowie weitere Bilder der Softwareoberfläche, können im Kapitel 5.1 "Erreichte Ergebnisse" nachgelesen werden.

Es wurde sichergestellt, dass alle Bestandteile der Software, ähnlich dem Baukastenprinzip, vorliegen. Diese Herangehensweise lässt es zu, dass auch künftig ohne große Anstrengung neue Inhalte innerhalb des Hauptcontainers platziert werden können. Der Anwender kann beliebig viele Konfigurationen im XML-Format Speichern und jederzeit wieder laden. Beim Beenden der Software kann der Zustand der Oberfläche automatisch gespeichert werden. In diesem Fall würde der Nutzer bei erneutem Starten die gewählte Einstellung wieder vorfinden, da die Standard XML-Datei beim Start wieder geladen wird.

3.4 Mehrsprachigkeit

3.4.1 Vorbetrachtungen

Es muss davon ausgegangen werden, dass der Übersetzer keine Erfahrungen mit der Syntax einer Programmiersprache oder der IDE besitzt, die bei der Programmierung verwendet wird. Aus diesem Grund muss auf eine saubere Trennung von Programmiertätigkeit und Übersetzungsarbeit geachtet werden. Ziel sollte es also sein, sämtliche Zeichenketten der Software, beispielsweise für Textfelder, Menüs, Tabellen und Meldungen, in externe Dateien auszulagern. Die Extensible Markup Language ist dafür bestens geeignet. Mit kurzen und selbst erklärenden Einträgen, den so genannten „Tags“, können sämtliche Unicode Zeichen deklarativ gespeichert werden. Das bedeutet, dass die Scriptsprache kein ausführbarer Code ist, was den Anspruch für den Übersetzer, bezogen auf die

Syntax, sehr gering hält. Das .Net Framework bietet für den Programmierer einen komfortablen Zugang zu diesen Dateien, um sie beispielsweise auslesen zu können. Darüber hinaus ist es möglich, mehrere Dateien auf ihre Konformität und Wohlgeformtheit hin zu überprüfen.

3.4.2 Lösungsansätze

Das .Net Framework stellt für die Realisierung von mehrsprachigen Anwendungen einen vorgefertigten Lösungsansatz zur Verfügung. Dieser weicht jedoch stark von dem Grundgedanken ab, der im Abschnitt 3.4.1 "Vorbetrachtungen" beschrieben wurde. Die Entwicklungsumgebung *Visual Studio* bietet mittels einer eingebetteten Funktion die Möglichkeit, alle bestehenden Fenster lokalisierbar zu machen. Dies bedeutet, es können beliebig viele Sprachen, "Kulturen" genannt, über ein Auswahlfeld ein- und umgestellt werden. Im Anschluss daran können die Text-Eigenschaften der Steuerelemente geändert werden. Die IDE legt im Hintergrund für jede, auf diesem Wege aktivierte Sprache, eine Ressource-Datei an. Darin sind die zu den Oberflächenobjekten gehörigen Zeichenketten gespeichert. Im Framework sind nun Funktionen enthalten, die ein einfaches Umschalten der lokalen Kultur erlauben.

Größter Nachteil dieser Lösung ist, dass der Übersetzer direkt in das Projekt eingreifen muss, um eine neue Sprache zu aktivieren. Man müsste zusätzlich per Hand die Text-Eigenschaft jedes Steuerelementes überschreiben. Dabei treten große Risiken auf. Der Übersetzer könnte einige Steuerelemente vergessen oder gar Fehler in das Softwareprojekt hinein bringen. Damit wäre die Stabilität der Anwendung nicht mehr gewährleistet. Davon abgesehen müsste der hier arbeitende Übersetzer auch Kenntnisse über die Entwicklungsumgebung und die Programmiersprache besitzen. Für das Forschungsprojekt ist diese Vermischung von Programmier- und Übersetzungstätigkeit keine Option.

Der einfachste Ansatz wäre nun, für jede angedachte Sprache eine XML-Datei anzulegen. Formal müssten sämtliche Dateien gleich aussehen und würden sich nur anhand der unterschiedlichen Spracheinträge unterscheiden. Seitens der Programmierung muss eine Verwaltungsinstanz, ein „LanguageManager“, geschaffen werden. Dieser muss in der Lage sein, die Spracheinträge zu laden und an das passende Steuerelement zu übergeben. Die zum Einsatz kommenden Sprachdateien sollten sich in einem separaten Ordner der Anwendung befinden. Dann könnte automatisch ermittelt werden, wie viele Sprachen für das Programm zu Verfügung stehen. Bei diesem Lösungsansatz gibt es jedoch noch eine Schwierigkeit. Es ist jene Tatsache, dass XML immer im Klartext gespeichert wird und dass jeder böswillige Anwender die Texte manipulieren oder löschen könnte. Um dieses Sicherheitsrisiko zu umgehen, müssten die Texte binär gespeichert werden.

Ein Ausbau der gesamten Sprachkomponente zu einer einzigen ausgelagerten Klassenbibliothek wäre ein großer Sicherheits- und Qualitätsgewinn. Alle Funktionalitäten wären von dem eigentlichen Softwareprojekt abgekoppelt und es könnten neue Sprachen nachinstalliert werden, ohne in die Anwendung eingreifen zu müssen. Man müsste nur eine neue Sprache in Form einer weiteren XML-Datei in der Sprachbibliothek aufnehmen. Die neue Version der DLL kann dann ohne Programmieraufwand, bei der unveränderten Applikation, aktualisiert werden. Damit die neu hinzu gekommene Sprache automatisch erkannt wird, müssen die verfügbaren Sprachen wieder dynamisch erkannt und ausgelesen werden. Die bereits angedachte "LanguageManager" Klasse für die Sprachdateien würde auch weiterhin ein Bestandteil der DLL bleiben.

Nach der Diskussion über die vorhandenen Lösungsansätze wird deutlich, dass nur eine individuelle Lösung entwickelt werden kann. Es ist notwendig, die Sprachdateien vor dem Nutzer zu verbergen. Die komfortable Erweiterung um

zusätzliche Sprachen sollte angestrebt werden. Im folgenden Abschnitt wird beschrieben, wie der durchdachte Lösungsansatz umgesetzt wurde.

3.4.3 Entwicklung und Einsatz einer Sprachbibliothek

In das neue Teilprojekt, was den Namen *Languages.dll* tragen soll, werden die Dateien für die sprachbezogenen Texte als Ressource eingebunden. Bei der Fertigstellung wird die Klassenbibliothek an das eigentliche Softwareprojekt angebunden. Die Sprachdateien besitzen den selbst definierten Dateityp "*.lang", damit man zweifelsfrei erkennt, dass es sich hierbei um eine Datei für landesspezifische Zeichenketten handelt. Der Dateiname muss der Landessprache entsprechen, die darin aufgeführt ist. Der Name ist beim Auslesen der vorhandenen Ressourcen wieder relevant.

Allgemein ist jede XML-Datei in einer Baumstruktur aufgebaut. Das „Wurzelelement“ ist allen anderen Tags übergeordnet. Es nimmt im Falle der Sprachdateien alle Knoten auf, die jeweils einem Fenster in der Applikation entsprechen. Die Tags für die verschiedenen Forms erfüllen nur die Funktion, dass sie die eigentlichen Spracheinträge kapseln und besser strukturieren. In der Ebene darunter gibt es nur noch eine Art von Tag. Diese beinhalten die Texte für die unterschiedlichen Landessprachen. Egal ob es sich um den Text eines Steuerelementes, einer Tabelle, oder eines Dialoges handelt, alle Einträge stehen in einem so genannten *ITEM*-Tag. Dieser besitzt ein Name-, ein Text- und ein ToolTipText-Attribut. Das Name-Attribut muss in der XML-Datei eindeutig sein und der Name-Eigenschaft des Steuerelementes innerhalb der Software entsprechen. Damit können die Zeichenketten, die im Text- und im ToolTipText-Attribut stehen, genau zugewiesen werden. Der Beispielcode 3-3 zeigt, wie die *LANG*-Dateien aufgebaut sind.

```
<APPLICATION>

    <FORM Name="Form-Name">

        <ITEM Name="Control-Name"
              Text="Text in einer Sprache"
              ToolTipText="ToolTip in einer Sprache" />

         <!-- jetzt folgen beliebig viele ITEM-Tags -->

    </FORM>

    <!-- jetzt folgen beliebig viele Fenster -->

</APPLICATION>
```

Quellcode 3-3 Struktur der *.lang Dateien

Damit nun auf die *ITEM*-Tags zugegriffen werden kann, muss die geforderte Ressource zunächst geladen werden. Die Entwicklung der Klasse *Language-Manager* genügte, um die Ressourcen auslesen und bereit stellen zu können. Zu Beginn des Lebenszyklus eines *LanguageManager* Objektes werden automatisch alle vorhandenen Sprachdateien erfasst und als öffentliche Liste bereit gestellt. Beim Starten des Simulator/Demonstrator wird diese Liste automatisch in einem Menüpunkt „Sprachen“ aufgeführt. Der *LanguageManager* lädt die vorerst eingestellte Standardsprache „deutsch“ und übergibt die deutschen Begriffe wie folgt. In der Methode *switchLanguage(String language, Form form)* benötigt die Managerinstanz die Sprache, in die gewechselt werden soll und die Form, bei der die Sprache umgestellt werden soll. Sobald das Name-Attribut eines FORM-Tags mit dem Namen des übergebenen Fensters übereinstimmt, werden alle *ITEM*-Tags, die zu dem Fenster gehören, in einer Liste zwischen gespeichert. Damit ist gewährleistet, dass auch nur die *ITEM*-Tags aus der Ressource geladen werden, die wirklich benötigt werden. Innerhalb der *switchLanguage()* Methode werden dann alle in der Form enthaltenen Steuerelemente rekursiv aufgelistet. Im Anschluss wird jedes Steuerelement durchlaufen und mit der Liste der *ITEM*-Knoten verglichen. Wenn eine Übereinstimmung vorliegt, dann wird der Text und, falls vorhanden, der ToolTipText aktualisiert. Diese Routine kann nun für alle Fenster der Anwendung durchlaufen werden.

Mit der Implementierung der *Languages.dll* in die Simulationssoftware wurde ein in sich geschlossener Ablauf realisiert. Die in der DLL enthaltenen Sprachen werden auf der Oberfläche aufgelistet und angezeigt. Durch den Automatismus ist jede ausgewählte Sprache zwangsläufig auch vorhanden und kann geladen werden. Dennoch gibt es zusätzlich eine Methode, die jene angeforderte Sprache auf ihre Existenz hin überprüft. Wird durch eine fehlerhafte Programmierung versucht, eine nicht valide Sprache zu laden, dann wird automatisch die Standardsprache geladen. Eine Fehlermeldung quittiert schließlich den fehlerhaften Versuch.

Nun gibt es noch Zeichenketten, die nicht statisch auf der Oberfläche angezeigt werden, sondern dynamisch geladen werden und sich während der Laufzeit verändern können. Dazu zählen Dialogfenster, Einträge in Tabellen und Statusmeldungen. Diese werden ebenfalls normal in einem *ITEM*-Tag gespeichert und erhalten dort einen eindeutiges Name-Attribut. Der *LanguageManager* bietet nun die Methode *getSingleEntry(String name)* an. Diese Funktion benötigt den Namen der zu dem gesuchten Text gehört und liefert ein String Array mit dem Text und ToolTipText zurück, welches dann eingesetzt wird.

3.4.4 Fazit zur Mehrsprachigkeit

Für die Zukunft ist es geplant, dass die Simulationssoftware zahlreiche Sprachen anbieten kann. Die Mehrsprachigkeit mittels einer Klassenbibliothek aus dem eigentlichen Softwareprojekt auszulagern, ist daher von großem Vorteil. Darüber hinaus befinden sich auch sämtliche sprachbezogene Texte in einer separaten *LANG*-Datei. Somit können weitere Fremdsprachen, ohne Risiko für die Programmierung, an externe Übersetzer weiter gegeben werden. Die Struktur der *LANG*-Dateien ist bewusst simpel gehalten, damit man sich bei der Übersetzung leicht zurecht findet. Die Textdateien mit den neuen Sprachen, müssen lediglich

als weitere Ressource eingebunden werden, da der *LanguageManager* in der Lage ist, diese vollkommen automatisch zu erkennen und auszulesen.

Die Implementierung der DLL in das Softwareprojekt des Simulators/Demonstrators ist mit nur einer benötigten Klasseninstanz ebenfalls überschaubar. Ein Umschalten der Sprache ist während der Laufzeit möglich, da jederzeit auf die in der Sprachbibliothek eingebetteten Ressourcen zugegriffen werden kann. Das Nachinstallieren einer neuen Sprache ist möglich, ohne an der eigentlichen Applikation etwas zu verändern. Auf diese Weise konnte erneut der Forderung nach einer modular aufgebauten Simulationssoftware mit hoher Flexibilität nachgekommen werden.

4 Die 3D-Visualisierungsbibliothek für drahtlose Sensornetze

4.1 Ausgangspunkt der Entwicklung

Die Darstellung von 3D-Grafiken unter Verwendung von OpenGL, soll ein wichtiges Merkmal des Simulators/Demonstrators werden. Damit kommt die Simulationsanwendung den modernen Ansprüchen der Computertechnologie nach und liefert nicht zuletzt eine wesentlich bessere Qualität für die Simulation drahtloser Sensornetze. Da die Simulations- und Demonstrationssoftware mittels der Programmiersprache C# entwickelt wird, ergibt sich das Problem der Zugreifbarkeit auf diese 3D-Grafikschnittstelle. Aus dem Kapitel 2.1 "Programmierung mit C#" ist bekannt, dass die OpenGL-Funktionalitäten nicht direkt über das .Net Framework erreicht werden können. Die Problematik soll über eine so genannte Wrapper-DLL gelöst werden. Diese spezielle Bibliothek spiegelt die OpenGL-API wider, indem sie deren Befehle mit jeweils einem analogen Funktionsaufruf umhüllt. Diese "gewrappten" Funktionen können wiederum mittels C# ausgeführt werden.

4.2 Einführung in OpenGL

Mit Hilfe der Programmierschnittstelle OpenGL kann man den 3D-Beschleuniger einer Grafikkarte nutzen und somit in Echtzeit beindruckende 3D-Grafiken darstellen. Implementiert wird die OpenGL-API durch die Grafikkartentreiber, die vom Hersteller der jeweiligen Grafikkarte mitgeliefert werden. Die Abbildung 4-1 zeigt ein Schichtmodell der OpenGL-Bibliothek. Die wichtigste darin enthaltene Bibliothek ist die *opengl32.dll*. Dem Programmierer stehen mit der Version 3.2 ungefähr 250 Befehle zur Verfügung[11]. Neben dieser gibt es noch weitere

[11] Quelle: http://de.wikipedia.org/wiki/OpenGL Zugriffszeit: 21.09.09

Bibliotheken, die dem Oberbegriff OpenGL zugeordnet werden können. Dazu zählt unter anderem die *Glu32.dll*, die so genannte *Utility Library.* Das GLU-Paket bietet fertige Algorithmen an, wodurch komplexere 3D-Objekte, beispielsweise Kugeln, mit nur einem Methodenaufruf dargestellt werden können. Die GLU-Funktionen basieren dabei vollständig auf den OpenGL-Grundfunktionen der *opengl32.dll*.

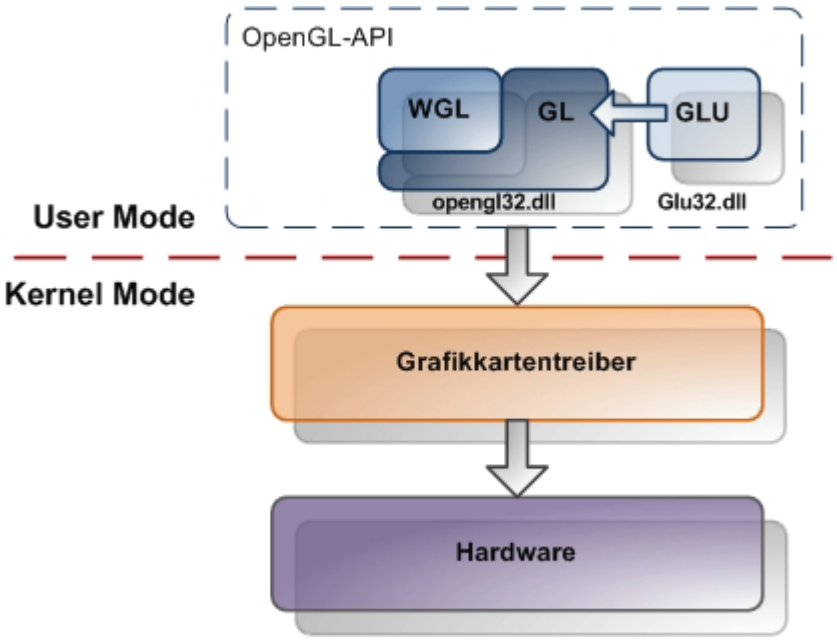


Abbildung 4-1 OpenGL Schichtmodell

Desweiteren gibt es noch Funktionen, die für verschiedene Betriebssysteme spezifisch sind. Das mit WGL bezeichnete Feld steht für eine Gruppe von Funktionen, die für Windows benötigt werden.

Da viele Unternehmen am OpenGL-Standard beteiligt sind, gibt es auch herstellerspezifische Erweiterungen, die dann ebenfalls zur OpenGL-API zählen können. Darauf soll in dieser Arbeit allerdings nicht weiter eingegangen werden. Mit der Abbildung 4-1 kann man sich auch den vertikalen Ablauf von der Software bis hin zur Hardware vorstellen. Für Programmiersprachen, wie beispielsweise C, C++ oder Delphi ist es möglich, die OpenGL-Schnittstelle direkt zu nutzen. Bei

diesen Sprachen genügt es also, einen Verweis auf die *opengl32.dll* anzugeben, damit die API in einem 3D-Programm eingesetzt werden kann. Für höher entwickelte Programmiertechnologien, wozu das .Net Framework und damit auch C# zählt, gibt es keinen mitgelieferten Zugang zur OpenGL-Grafikschnittstelle. Für das Microsoft Framework soll mit dieser Arbeit eine selbst zu entwickelnde und zu realisierende Zugangsmöglichkeit geschaffen werden.

OpenGL als Zustandsmaschine

Um den Ablauf eines OpenGL-Rendervorgangs besser verstehen zu können, hat es sich allgemein durchgesetzt, dass man sich für OpenGL eine Art "Zustands-maschine" vorstellt. Bei dieser OpenGL-Maschine gibt es eine Vielzahl von Parametern, die eingestellt werden können. Darunter zählen unter anderem die Farbe, Transparenzeigenschaften, Lichtobjekte oder Texturen. Die getroffenen Einstellungen beeinflussen alle Zeichenobjekte, die anschließend gerendert werden sollen. Wenn man zu Beginn des Zeichenvorgangs eine Einstellung für einen Parameter trifft, dann wird diese Einstellung solange auf alle nachfolgend zu zeichnenden Objekte angewendet, bis die Einstellung verändert wird. Die Arbeitsweise von OpenGL als Zustandsmaschine erleichtert dem Programmierer den Umgang mit der Grafikschnittstelle enorm, denn man muss nicht bei jeder Zeichenoperation eines Körpers alle Parametereinstellungen ständig mit übergeben. Es genügt, die OpenGL-Maschine zu Beginn zu konfigurieren und nur bei Bedarf einige der Parameter zu verändern.

Der Zeichenvorgang

Im folgenden Beispiel soll ein buntes Dreieck entstehen, um den Anwendungs-ablauf mit OpenGL zu verdeutlichen und um einige OpenGL spezifischen Begriffe kennen zu lernen. Mit den OpenGL-Grundfunktionen lassen sich nur Primitive

zeichnen. Dazu zählen Punkte, Linien, Dreiecke, Quadrate und Polygone mit mehr als vier Eckpunkten. Die Anlage A.4 "OpenGL Rendermodus" zeigt, welche Zeichenmodi es gibt, welche an die *glBegin()* Methode übergeben werden können und zu welchem Zeichenergebnis diese führen. Wie man daran sieht, bestehen Zeichenbefehle nur aus einer Vielzahl von Anweisungen zum Platzieren eines Vertex. Diese müssen immer von einer *glBegin()* und *glEnd()* Methode umschlossen werden. Der Aufruf von *glBegin()* benötigt allerdings noch den Übergabeparameter *modus*. Dieser "Rendermodus" definiert, ob die Summe der Punkte einzeln dargestellt, zu Linien verbunden oder als Flächen gerendert werden sollen.

Der Farbparameter der OpenGL-Maschine kann, wie bereits erwähnt, jederzeit verändert werden. Auf diese Weise lassen sich Farbverläufe leicht erzeugen, indem man vor dem Eckpunkt einer Fläche die Farbeinstellung verändert. Die Farbwerte zwischen den Punkten werden dann interpoliert. Der Rendermodus *GL_TRIANGLES* definiert, dass aus allen folgenden Vertices dreieckige Flächen erstellt werden sollen. Das Beispiel in Abbildung 4-2 zeigt, wie ein buntes Dreieck beschrieben wird, was fünf Einheiten in die Bildebene hinein geschoben wurde.

```
glBegin(GL_TRIANGLES);

glColor3f(1, 0, 0);
glVertex3f(1, 1, -5);

glColor3f(0, 1, 0);
glVertex3f(3, 1, -5);

glColor3f(0, 0, 1);
glVertex3f(2, 3, -5);

glEnd();
```

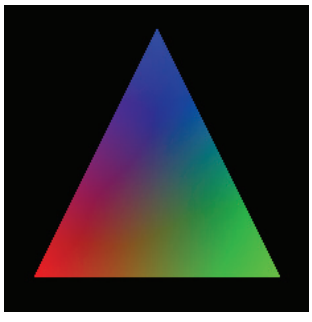


Abbildung 4-2 OpenGL Dreieck

OpenGL Syntax

Der kurze Beispielcode von Abbildung 4-2 wurde von allen unnötigen Befehlen bereinigt und soll nur exemplarisch den Zeichenvorgang verdeutlichen. Es soll gezeigt werden, wie die gesamte Syntax bei OpenGL strukturiert ist. Das Befehlsschema ist bei allen eingesetzten Grafikbibliotheken aus dem OpenGL-Paket immer gleich. Reine OpenGL-Funktionen erkennt man an dem Präfix „gl“. Die *OpenGL Utility Library* wird beispielsweise mit „glu“ abgekürzt, um die Herkunft der Anweisungen unterscheiden zu können. Diese Vorsilben stehen immer vor der eigentlichen Anweisung. Anschließend, mit Großbuchstabe fortführend, folgt der Befehl. Ihm schließt sich die Anzahl und der Datentyp der benötigten Übergabeparameter an. Für das Dreieck werden demnach sowohl bei der Farbzuweisung, als auch bei der Platzierung der Eckpunkte, drei Gleitkommazahlen vom Typ *float* benötigt. Bei dem Funktionsaufruf *glColor3f()* erwartet die OpenGL-Maschine drei Werte zwischen Null und Eins, die für die Intensität der Farbkanäle Rot, Grün und Blau stehen. Die Funktion *glVertex3f()* beschreibt einen Punkt im Koordinatensystem und benötigt dafür die X-, Y- und Z-Position des Punktes. In der Tabelle 4-1 sind die möglichen Datentypen aufgelistet, die OpenGL entgegen nehmen kann.

Suffix von OpenGL Funktionen	**Datentyp**
i	Integer
f	Float
d	Double
b	Byte
s	Short
us	vorzeichenloser Short
ui	vorzeichenloser Integer

Tabelle 4-1 OpenGL Datentypen

Viele Funktionen existieren noch in einer Variante, bei der ein *Array* eines bestimmten Datentyps als Übergabeparameter erwartet wird. In Diesem Fall schließt sich ein „v“ an den Datentyp an. Das „v“ steht an dieser Stelle für "Vektor" und gibt an, dass ein *Array* mit der betreffenden Anzahl an Parametern erwartet wird.

Theoretisch wird jedes beliebig komplexe Objekt aus den OpenGL-Primitiven erstellt. Es wäre jedoch ein viel zu großer Aufwand, komplexere Körper, wie beispielsweise Kugeln, Zylinder oder Kegel, immer in einem selbstständig erstellten Algorithmus zu erzeugen. Da die OpenGL-Maschine nur Befehle besitzt, um Primitive zu rendern, wurde die *OpenGL Utility Library* zur API hinzugefügt. Mit der GLU ist es möglich, auch komplexere Grundkörper mit einem einzelnen Funktionsaufruf darzustellen. Dahinter verbirgt sich allerdings ebenfalls nur eine Routine aus Zeichenbefehlen, welche die bekannten Primitiven verwendet. Die *Glu32.dll* setzt also auf den OpenGL-Grundfunktionen auf und ist daher ein fester Bestandteil des gesamten OpenGL-Pakets.

4.3 Anforderungsanalyse

Es soll eine Klassenbibliothek designt und realisiert werden, mit der die nativen OpenGL-Kommandos unter C# verwendet werden können. Diese DLL muss in einem Softwareprojekt eingebunden werden, um die Weiterleitung, von Befehlen zur 3D-Grafik Darstellung, an die OpenGL-Maschine durchführen zu können. Diese Aufgabe stellt die größte Herausforderung für diese Arbeit dar.

Bevor man mit der Entwicklung einer solch umfangreichen Klassenbibliothek beginnen kann, muss man einige Besonderheiten kennen und berücksichtigen. Die Visualisierungsbibliothek soll in erster Linie für die Simulation von drahtlosen Sensornetzen eingesetzt werden. Die detaillierten Anforderungen, die an diese

Visualisierungsbibliothek gestellt werden, sollen in diesem Abschnitt aufgeführt und erläutert werden.

Design als Wrapper-DLL

Damit die Funktionen aus der OpenGL-Programmierschnittstelle in einer "höher entwickelten" Programmiersprache verwendet werden können, muss jeder einzelne Befehl umhüllt, also „gewrappt“ werden. Ziel ist es also eine Klassenbibliothek zu erstellen, die ausgewählte OpenGL-Befehle aufnimmt und für C#-Projekte bereit stellt. Mit dieser selbst entwickelten Bibliothek soll eine Brücke zwischen der Programmiersprache C# und der eigentlichen OpenGL-API geschlagen werden. Die Wrapper-DLL soll den Namen *SharpOGL_AS.dll* tragen. Damit wäre bereits im Namen "SharpOGL" der neuen DLL der Bezug zu C# hergestellt. Der Suffix "AS" steht für die Initialen des Autors. Die entstehende Visualisierungsbibliothek soll sich neben der *opengl32.dll*, auch auf die *Glu32.dll* und die Funktionen, die abhängig vom Betriebssystem Microsoft benötigt werden, stützen.

Übersichtlichkeit und Eindeutigkeit

Ein übersichtlicher Aufbau der DLL ist eine weitere Grundvoraussetzung, wenn die Wrapper-DLL erfolgreich eingesetzt werden soll. Ziel soll es sein, ein möglichst exaktes Abbild der nativen OpenGL-Grafikschnittstelle zu erhalten. Die Methoden, die unter C# Verwendung finden, dürfen darüber hinaus nicht exakt so heißen wie die originalen Befehle, da dies zwangsläufig zu einem Konflikt führen würde. Erschwerend kommt hinzu, dass nur Fachliteratur zur Verfügung steht, die sich mit den originalen OpenGL-Befehlen befasst. Da es sich bei der C# Spezifikation um eine Eigenentwicklung handelt, muss also nach einer Befehlssyntax für die C# Methoden gesucht werden, die dem zu Grunde liegenden Befehl

eindeutig zugeordnet werden kann. Die neue Schreibweise muss also einen Unterschied aufweisen, darf aber dabei den eigentlichen OpenGL-Aufruf nicht entfremden.

Anpassung an die Entwicklungsumgebung

Die Arbeit mit *Visual Studio* nimmt dem Programmierer eine Vielzahl an Schreibarbeit ab. Viele Dinge, im Bezug auf die Oberflächenprogrammierung, können über die IDE eingestellt werden, anstatt mühsam programmiert zu werden. Im Hintergrund laufen dann Automatismen ab, die den Quellcode generieren. Dieser Komfort soll bei der Anwendung der Wrapper-DLL nicht verloren gehen. Es wird also gefordert, dass man bereits durch die Entwicklungsumgebung *Visual Studio* bei der Anwendung der selbst entworfenen Bibliothek, Unterstützung im Bezug auf einige Eigenschaften erhält. Die hier angesprochenen Eigenschaften sollen in einem separat entwickelten Steuer-element, dem *SceneGraph,* vorhanden sein.

Implementierung eines SceneGraphen

Es ist im Hinblick auf die Arbeit mit *Visual Studio* angedacht, ein Steuerelement zu entwerfen, welches dann von der neuen Klassenbibliothek mitgeliefert wird. Auf der Oberfläche des so genannten *SceneGraphs* findet dann die 3D-Darstellung statt. Dieser kann dann wie gewohnt über die Toolbox auf eine freie Fläche eines Fensters gezogen werden. Ein weiterer Vorteil wäre, dass dieses Steuerelement die Initialisierung des Rendervorgangs übernehmen könnte. Dieser komplexe Vorgang soll dann automatisch bei der Instanziierung des *SceneGraphen* geschehen. Bei der späteren Anwendung der Klassenbibliothek könnte das manuelle Einrichten der OpenGL-Maschine komplett entfallen. Damit soll die

eigentliche 3D-Grafik Programmierung für Einsteiger komfortabler werden und schneller zu erlernen sein.

Das Steuerelement soll mit speziellen Eigenschaften versehen werden, die sich über die IDE konfigurieren lassen. Die *FrameRate*, also die Anzahl der zu zeichnenden Bilder pro Sekunde, ist die wichtigste Eigenschaft. Mit ihr soll sich die Anzahl der Renderzyklen in einer Sekunde begrenzen lassen und somit ressourcenschonend arbeiten. Eine weitere sinnvolle Eigenschaft wäre eine Statusleiste, die sichtbar gemacht oder verborgen gehalten werden kann. In dieser können wichtige Informationen über den Rendervorgang ausgegeben werden. Darüber hinaus soll der Anwender der DLL die Hintergrundfarbe bereits beim Design einer 3D-Anwendung einstellen können.

Hinzu kommt, dass man diesem Steuerelement *Events* zuweisen können muss, die dann genutzt werden sollen. Das wichtigste Ereignis ist das „Paint-Event“. Dieses soll entsprechend der *FrameRate* ausgeführt werden. Der 3D-Grafik Programmierer muss dieses Ereignis unbedingt implementieren, damit ein Zeichenvorgang stattfinden kann. Alle in diesem Event stehenden Befehle werden dann zyklisch als Hauptschleife ausgeführt. Ein ebenfalls sehr wichtiges Ereignis soll eintreten, wenn sich die Größe des Steuerelementes ändert. Denn damit könnte sich das Seitenverhältnis der Ausgabeoberfläche verändern, was einen großen Einfluss auf die OpenGL-Maschine hätte. Der Softwareentwickler, der mit der Visualisierungsbibliothek arbeitet, muss die Möglichkeit haben, das „Resize Event“ auf seine Bedürfnisse anzupassen. Weitere nützliche Ereignisse sollen der Interaktion mit dem Nutzer dienen. Darunter fallen alle Events, die mit der Maus oder der Tastatur ausgelöst werden können. Mit Hilfe dieser „Mouse und Key Events“ ist der Weg bereitet, eine virtuelle Umgebung zu programmieren, in der sich der Anwender einer 3D-Software frei bewegen kann.

Auswahl der implementierten Befehle

Wie bereits erwähnt, besitzt man mit OpenGL ein mächtiges Werkzeug für die 3D-Grafik Programmierung. Die Fülle an Funktionen und Konstanten sollen in der knappen Entwicklungszeit nicht vollständig bearbeitet werden. Die wichtigsten Befehle zum Erstellen einer virtuellen Umgebung müssen jedoch vorhanden sein. Darunter zählen das Zeichnen sämtlicher Primitive, Farbeinstellungen, Lichtobjekte und die wichtigsten Funktionen zum Konfigurieren des Rendervorgangs. Aufbauend auf der damit vorhandenen Struktur, lässt sich die DLL später nach Belieben erweitern. Es soll auch bedacht werden, dass weitere Bibliotheken, die zur OpenGL-Familie gehören, zumindest teilweise eingebunden werden müssen. Exemplarisch sollen einige GLU-Funktionen implementiert werden. Dabei ist darauf zu achten, dass bei der Struktur der Eigenentwicklung eine saubere Trennung zwischen GL- und GLU-Funktionen eingehalten wird. Ziel der Einbindung der *Utility Library* soll es sein, ein vereinfachtes Rendern von Grundkörpern, wie Kugeln oder Zylindern, zu ermöglichen.

Objektorientiertheit

Bei OpenGL handelt es sich um statische Funktionsaufrufe ohne objektorientiertes Verhalten, wie die Abbildung 4-2 gezeigt hat. Für die Entwicklung unter C# soll jedoch eine objektorientierte Lösung angestrebt werden. Grundvoraussetzung ist demzufolge, dass für die Visualisierungsbibliothek nicht statische Klassen geschrieben werden, die dann in einem anderen Softwareprojekt instanziiert werden können. Der Einsatz von verschiedenen Klassen macht Überlegungen zur deren Sichtbarkeit von außen auf die DLL erforderlich. Der Entwickler, der sich später mit der 3D-Visualisierungsbibliothek befasst, soll nach Möglichkeit nur über eine einzige Instanz Zugriff auf alle bereit stehenden Funktionen erhalten. Damit soll gewährleistet werden, dass es zu keinem

inkonsistenten Zustand während der Laufzeit kommen kann. Weiterhin wird dadurch die Arbeit mit der Wrapper-DLL sicherer gestaltet, da Bestandteile, die zur Initialisierung genutzt werden, nur intern und automatisiert aufgerufen werden können und damit nicht fehlerhaft eingerichtet werden können.

4.4 weitere Lösungskonzepte

Es gibt bereits Lösungsansätze, welche die 3D-Grafikschnittstelle OpenGL im .Net Framework verfügbar machen, die prinzipiell eingesetzt werden könnten. Die Eigenschaften dieser Bibliotheken könnten jedoch von den erarbeiteten Anforderungen zum Teil stark abweichen. Auf den folgenden Seiten sollen zwei Bibliotheken näher untersucht werden.

CsGL

Die erste OpenGL-Bibliothek für C#, die vorgestellt werden soll, nennt sich *CsGL* [CsGL 03]. Ihre Weiterentwicklung wurde im April 2003 eingestellt, wodurch sie mittlerweile veraltet ist. Auf der Internetpräsenz des Entwicklers wird geschrieben, dass die Bibliothek "stabil und gut genug für die meisten Applikationen" arbeitet, aber dass sie "noch einige bekannte Probleme"[12] hat. Die mitgelieferten Quellcodebeispiele haben gezeigt, dass bei *CsGL* mit statischen und abstrakten Klassen gearbeitet wurde. Des weiteren gibt es keine Struktur, die mit einem *SceneGraphen* vergleichbar wäre, wodurch die Arbeit mit OpenGL in keiner Weise vereinfacht wird. Die *CsGL*-Bibliothek diente seinerzeit also nur der exakten Übertragung von OpenGL-Befehlen in das .Net Framework.

[12] siehe [CsGL 03]

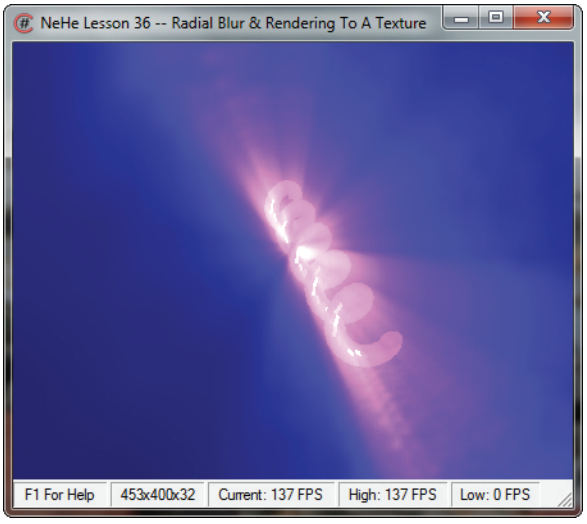


Abbildung 4-3 NeHe Helix

Abbildung 4-3 zeigt eine Software, die eine Helix animiert. Der Algorithmus zum Darstellen der Helix, wurde mit der *CsGL*-Bibliothek erstellt. Unter Verwendung von *CsGL* lässt sich diese relativ anspruchsvolle Animationen mit hoher Geschwindigkeit rendern. Die Performanz der Bibliothek scheint also für leistungskritische Anwendungen zu genügen. Im Abschnitt 5.3 "Ausblick auf Erweiterungen der Visualisierungsbibliothek" wird noch einmal ein Leistungsvergleich mit der *CsGL*-Bibliothek angesprochen.

OpenTK

Eine weiteres Konzept, die *Open Toolkit Library* [TK 09], wurde im Oktober 2009 auf der offiziellen OpenGL-Webseite [OGL 09] präsentiert. *OpenTK* ist ebenfalls ein Open Source[13] Projekt, womit man OpenGL gestützte 3D-Inhalte mit C# programmieren kann. Diese Bibliothek befindet sich momentan noch in der Entwicklungsphase.

Die *OpenTK*-Bibliothek unterscheidet sich in ihrem Aufbau stark von der geplanten Struktur der *SharpOGL_AS.dll*. Es ist aufgefallen, dass das *OpenTK*

[13] Software, deren Quellcode sowie deren Verwendung ohne Einschränkungen zur Verfügung steht

Projekt sehr hohe Systemanforderungen besitzt, da das Ausführen der mitgelieferten Beispiele, auf etwas älterer Hardware kaum möglich gewesen ist. *OpenTK* besitzt im Gegensatz zu *CsGL* ein Steuerelement, welches einem *SceneGraphen* ähnlich ist. Neben der reinen 3D-Grafik Programmierung wird bei dem *OpenTK*-Projekt auch an Komponenten für OpenAL[14] und OpenCL[15] Schnittstellen gearbeitet. Dadurch wird deutlich, dass mit diesem Projekt vordergründig Multimedia- oder Spiele-Entwickler angesprochen werden. Das hat zur Folge, dass hier ganz andere Schwerpunkte bei den Funktionalitäten gesetzt werden, als für diese Abschlussarbeit herausgearbeitet wurden. Zudem wurden die meisten OpenGL-Befehle zu komplexeren Funktionen zusammen gefasst, was eine erneute Einarbeitungsphase in die *OpenTK*-Syntax nach sich ziehen würde. Durch die neu geschaffene Syntax wird keine direkte Referenz auf die OpenGL-Schnittstelle geliefert, sondern, wie der Name schon verrät, ein "Toolkit" für die Darstellung 3D-Grafiken angeboten.

Fazit

Die Notwendigkeit der Eigenentwicklung der Visualisierungsbibliothek ist, trotz des Vorhandenseins anderer, fertiger Lösungen, unumstritten. Für den geplanten langfristigen Einsatz, bietet nur der Aufbau von eigenen Kompetenzen die notwendige Sicherheit und Kenntnisse über die Technologie. Bei Verwendung kostenloser Bibliotheken, kann eine stetige Weiterentwicklung nie garantiert werden. Hinzu kommt, dass die technische Unterstützung bei Fehlfunktionen nicht gewährleistet ist. Die bestehenden Open Source Lösungen sind zudem nicht exakt auf die Anforderungen des Simulator/Demonstrator zugeschnitten. Im folgenden Abschnitt wird die Entwicklungsphase der OpenGL-Visualisierungs-bibliothek ausführlich beschrieben.

[14] Open Audio Library
[15] Open Computing Language

4.5 Entwicklung der Visualisierungsbibliothek

4.5.1 Strukturierung der Visualisierungsbibliothek

Die Abbildung 4-4 soll einmal verdeutlichen, wo die Visualisierungsbibliothek einzuordnen ist. Wie man anhand des grün gestrichelten Rahmens sieht, besitzen Anwendungen, die mit Hilfe des .Net Frameworks entstehen, zunächst keinen direkten Zugang zur OpenGL gestützten 3D-Grafik Programmierung. Erst mit der Anbindung der zu entwickelnden *SharpOGL_AS.dll* ist der geplante Zugriff auf die OpenGL-Maschine möglich.

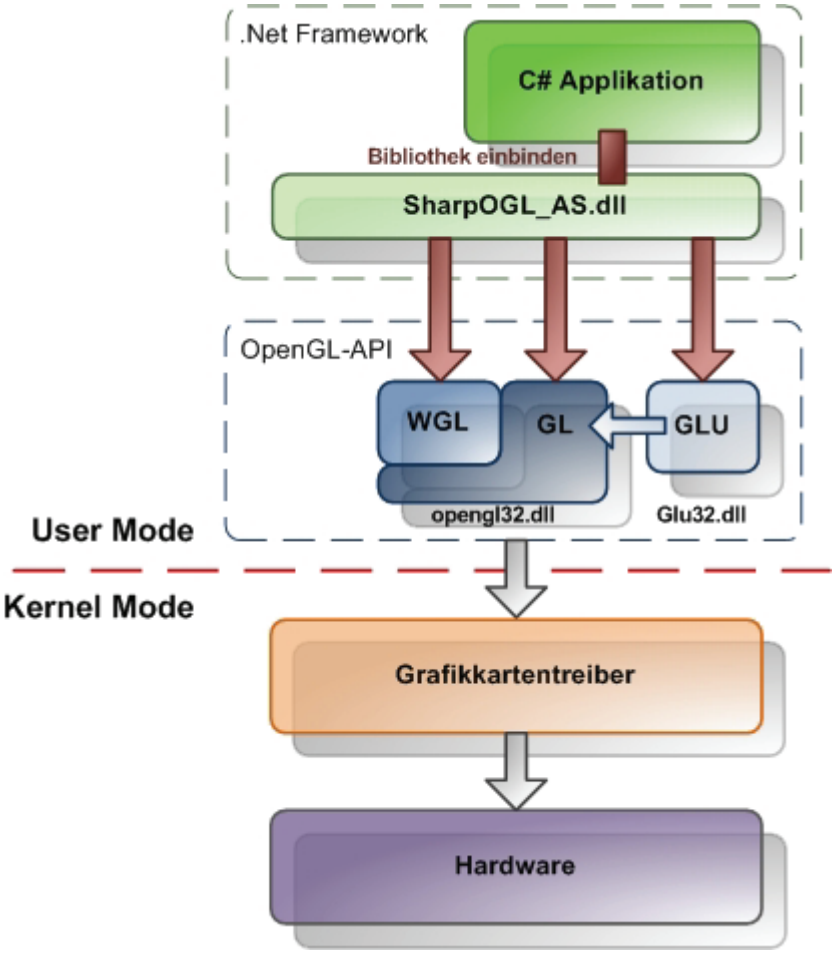


Abbildung 4-4 C# OpenGL Schichtmodell

Über die neue Visualisierungsbibliothek erhält man letztlich Zugriff auf alle übersetzten Bestandteile der OpenGL-API. Innerhalb der Bibliothek gibt es zu jeder Komponente des OpenGL-Paketes eine eigene Schnittstelle. Nachdem die Befehle an die OpenGL-Maschine weiter gegeben wurden, läuft der gleiche Prozess ab, wie beispielsweise bei einer C++ Anwendung, die direkt mit OpenGL kommuniziert.

Die selbständig entwickelte C#-Bibliothek besteht aus den in Abbildung 4-5 abgebildeten Komponenten, die für die Zusammenarbeit mit OpenGL erforderlich sind. Dieses UML-Klassendiagramm[16] spiegelt nur die wichtigsten Abhängigkeiten innerhalb der *SharpOGL_AS.dll* wider und stellt nicht den vollständigen Funktionsumfang dar. Wenn die Wrapper-DLL verwendet wird, ist nur die Komponente *SceneGraph*, die Klasse *OpenGL*, sowie die statische Klasse *GL_DEF* aus dem Namensraum *Definitions* sichtbar. Diese drei Bestandteile sind für die unmittelbare Anwendung der Bibliothek entwickelt worden. Die restlichen Komponenten der DLL können nur über die sichtbaren Klassen erreicht werden, oder sind für den Anwender der Visualisierungsbibliothek nicht von Bedeutung. Auf den kommenden Seiten soll das in Abbildung 4-5 dargestellte UML-Diagramm ausführlich beschrieben werden.

[16] Unified Modeling Language

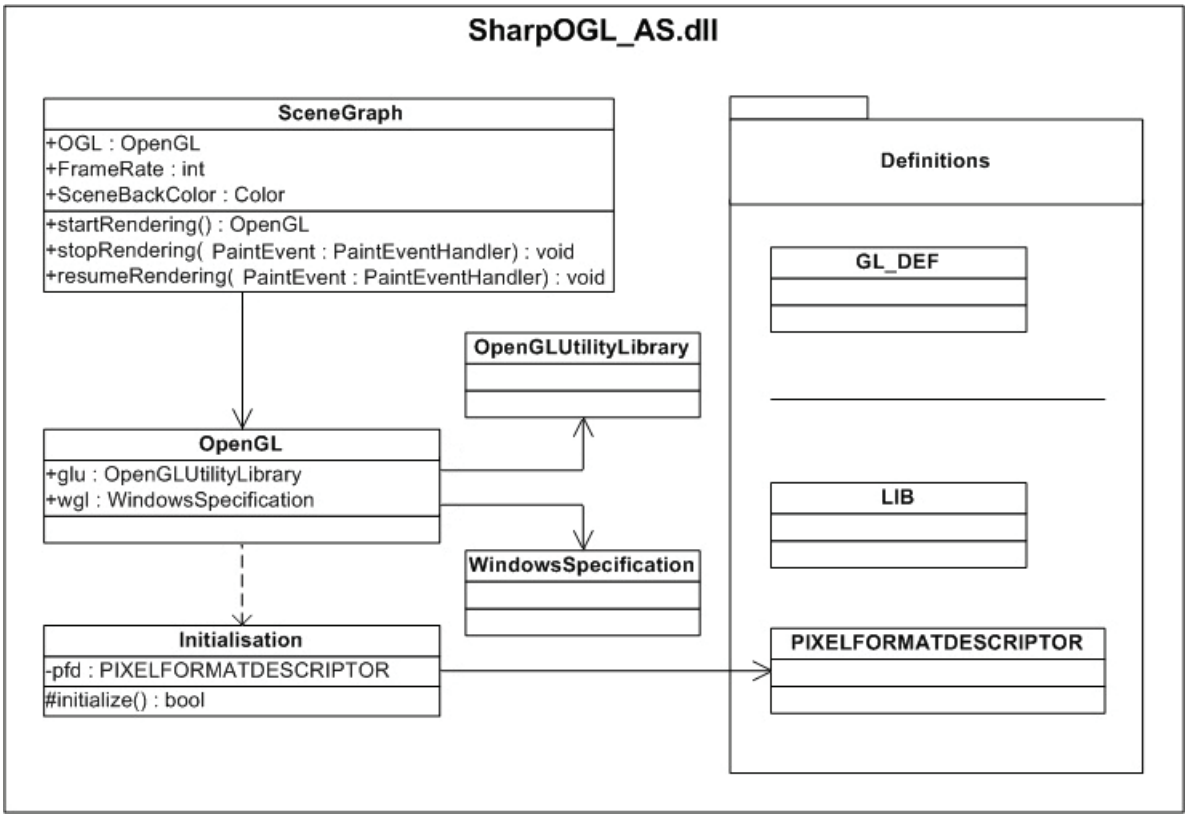


Abbildung 4-5 UML-Klassendiagramm von SharpOGL_AS.dll

Klasse - Initialisation

Die Klasse *Initialisation* wurde nur für den Initialisierungsvorgang entworfen und wird an die *OpenGL* Klasse vererbt. Der Vorgang der Initialisierung koppelt die OpenGL-Maschine an ein Ausgabemedium an. In diesem Fall der *SceneGraph*, dessen Instanz auf dem Monitor zu sehen ist. Bei den Vorbereitungen auf den Rendervorgang muss man der OpenGL-Zustandsmaschine unter anderem mitteilen, ob eine doppelte Pufferung des Ausgabebildes stattfinden soll, oder wie viel Speicher pro Bildpunkt reserviert werden soll.

statische Klasse - PIXELFORMATDESCRIPTOR

Die Klasse *PIXELFORMATDESCRIPTOR* ist nur intern zugänglich, da sie für den Gebrauch der *SharpOGL_AS.dll* keine Rolle spielt. Sie ist allerdings für den

Initialisierungsvorgang unentbehrlich. Dieser läuft vollständig innerhalb der Bibliothek ab und daher gibt es von außen, auch aus Sicherheitsgründen, keinen Zugang. In dieser Klasse werden 26 Parameter eingestellt, die für die Anzeige eines Bildpunktes von Bedeutung sind. Darunter zählen Informationen über die Farbtiefe, die einzelnen Farbkanäle und diverse Puffereigenschaften zum Zwischenspeichern der Pixeldaten. Beim Initialisieren der OpenGL-Maschine wird dieser *PIXELFORMATDESCRIPTOR* benötigt und muss im Voraus eingestellt worden sein.

Klasse - SceneGraph

Durch die objektorientierte Denkweise ist das Steuerelement *SceneGraph* vorgesehen. Mit dessen Methode *startRendering()* wird der Rendervorgang gestartet. Mit diesem einzigen Aufruf wird automatisch ein Objekt der Klasse *OpenGL* instanziiert und initialisiert. Die als Rückgabetyp ausgegebene *OpenGL* Instanz macht dann alle nativen OpenGL-Befehle zugänglich. Nach dem Aufruf der *startRendering()* Methode wird das *Paint*-Ereignis des *SceneGraphs* standard-mäßig 60mal in der Sekunde durchlaufen. Ab diesem Zeitpunkt kann der 3D-Grafik Programmierer das fertig initialisierte *OpenGL* einsetzen. Nur mit dem übergebenen *OpenGL* Objekt, kann man alle implementierten Funktions-anweisungen innerhalb des *Paint*-Events ausführen und somit auf den Monitor übertragen. Über die Methoden *stopRendering()* und *resumeRendering()* kann man während der Laufzeit die 3D-Ausgabe stoppen und wieder weiter laufen lassen. Dazu muss man den *PaintEventHandler*, der die Zeichenoperationen ausführt, an die *stopRendering()* und *resumeRendering()* Methode übergeben. Dadurch kann eine Ab- beziehungsweise Anmeldung des *Paint*-Events innerhalb des SceneGraphs stattfinden. Nach der Abmeldung des Paint -Ereignisses findet keine Beanspruchung des Computersystems durch die 3D-Visualisierung statt.

Klasse - OpenGL

Allein in der Klasse *OpenGL* befindet sich die Sammlung von nativen OpenGL-Befehlen, die über einen „DLL Import“ der originalen OpenGL-API hinzu geladen werden müssen. Die importierten "echten" OpenGL-Befehle sind nach außen hin nicht sichtbar. Diese Funktionen werden durch eine beinahe identisch klingende Methode umhüllt. Wie dies geschieht, zeigt der kurze Beispielcode 4-1. Auf diese Weise sind sämtliche Methoden, die bereits übersetzt wurden, gestaltet. Um der Anforderung gerecht zu werden, dass die neuen Methoden ihren Originalen eindeutig zugeordnet werden können, wurde bei dem Namen lediglich auf den Präfix „gl“ verzichtet. Damit wird ein Namenskonflikt verhindert, denn aus *glBegin()* wird *Begin()*. An dieser Stelle muss dem 3D-Grafik Programmierer unbedingt folgendes empfohlen werden. Das *OpenGL* Objekt, welches zum Einsatz kommt, sollte mit „gl“ bezeichnet werden. Damit würde der Methodenaufruf *gl.Begin()* fast exakt wie sein statisches Vorbild aus der OpenGL-Maschine aussehen.

```
// benötigter Namensraum für die Einbindung von nicht-verwaltetem Code
using System.Runtime.InteropServices;

// Ausschnitt aus der SharpOGL_AS.dll, erbt von Initialisation
public class OpenGL : Initialisation
{
    // sichtbarer Methodenaufruf
    public void Begin(uint mode)
    {
            // wrappt die native OpenGL Funktion
            glBegin(mode);
    }

    // nicht sichtbarer Methodenaufruf
    [DllImport("opengl32.dll")]
    protected static extern void glBegin(uint mode);
}
```

Quellcode 4-1 Übersetzung der OpenGL-Methode *glBegin()*

Klasse - OpenGLUtilityLibrary

Die Klasse *OpenGLUtilityLibrary* ist analog der *OpenGL* Klasse aufgebaut. Sie beinhaltet die nativen GLU-Befehle und kapselt diese in einer Methode, die ebenfalls nur um den Präfix bereinigt wurde. Es würde für diese Klasse keinen Sinn ergeben, eine separate Initialisierung durchzuführen. Aus Sicht der OpenGL-Maschine gibt es ohnehin nur ein Ausgabemedium. Hinzu kommt, dass auch die GLU-Befehle innerhalb der *Glu32.dll* statisch aufgerufen werden. Anhand der Abbildung 4-5 kann man erkennen dass die Klasse *OpenGL* bereits eine Eigenschaft *glu* besitzt. Es ist also völlig ausreichend, wenn man über diese Eigenschaft auf die Methoden der *OpenGLUtilityLibrary* Klasse zugreift. Für den Aufruf einer GLU-Funktion würde sich ein ähnliches Bild, wie bei einem Aufruf eines GL-Befehls ergeben. Unter der Voraussetzung, dass die *OpenGL* Instanz mit „gl“ bezeichnet wurde, würde die Methode zum Zeichnen einer Kugel *gl.glu.Sphere()* lauten. Wie man sieht, lassen sich die GLU-Funktionen leicht über das *OpenGL* Objekt erreichen.

Klasse - WindowsSpecification

Für Windows spezifische Funktionen gilt die gleiche Herangehensweise, wie für die *OpenGLUtilityLibrary*. Die Klasse *WindowsSpecification* enthält alle Funktionen, die den Präfix „wgl“ tragen. Ein Teil dieser Befehle ist notwendig um die OpenGL Initialisierung korrekt durchführen zu können. Allerdings gibt es auch Methoden, die für den Rendervorgang bestimmt sind. Ein Beispiel wäre das Zeichnen von zweidimensionaler Schrift, die in sämtlichen *Font*[17] Typen erscheinen kann. Die Eigenschaft *wgl* der Klasse *OpenGL* macht den Zugriff auf diese Funktionen möglich.

[17] Bezeichnung für Schriftarten unter Windows

statische Klasse - GL_DEF

Des weiteren wurde noch ein Namensraum eingeführt, um die Visualisierungsbibliothek besser strukturieren zu können. In dem Paket *Definitions* befindet sich zunächst nur eine sichtbare, statische Klasse *GL_DEF*. Diese beinhaltet nur konstante Werte. Die Klasse wird benötigt, um alle von der OpenGL-Maschine erwarteten Einstellungsparameter zu speichern. Bereits bekannte Parameter, wie *GL_TRIANGLES* zum Einstellen des Rendermodus, sind darin zu finden. Es gibt noch eine Menge weiterer Konstanten, die darin gespeichert sind, beispielsweise Parameter für Lichtobjekte, Texturen oder Transparenzeinstellungen.

statische Klasse - LIB

Für den Anwendungsprogrammierer ist die interne statische Klasse *LIB* ohne Bedeutung. Sie dient nur der zentralen Verwaltung von Zeichenketten, die innerhalb der Visualisierungsbibliothek benötigt werden. Es werden beispielsweise an vielen verschiedenen Stellen innerhalb der DLL, die Bezeichnungen der nativen OpenGL-Bibliotheken *opengl32.dll* oder *Glu32.dll* benötigt, die hier in *LIB* abgelegt worden sind.

4.5.2 Besonderheiten gegenüber nativem OpenGL

Die Arbeit mit dem .Net Framework von Microsoft bietet einige Annehmlichkeiten, die bereits bei der Strukturierung der Visualisierungsbibliothek beachtet wurden. Um einen komfortablen und dennoch effizienten Zugang zur 3D-Grafik Programmierung mit OpenGL herzustellen, wurde ein einsatzbereites Steuerelement entworfen. Ziel ist es, den .Net Entwicklern den Einstieg in die Welt von OpenGL zu erleichtern, da diese meist unerfahren im Umgang mit dieser hardwarenahen Programmierlogik sind.

Eine grundlegende Abweichung zur OpenGL-Befehlsstruktur ist die geänderte Schreibweise der Funktionen. Neben dem bereits bekannten Verzicht auf den Präfix bei OpenGL-Anweisungen, gibt es noch eine weitere Besonderheit. Um die Lesbarkeit von Methoden zu verbessern, die einen oder mehrere Übergabeparameter erwarten, wurden diese Zeichen durch einen Bodenstrich vom eigentlichen Befehl getrennt. Der Aufruf von *glColor3f()* hat sich also bei der C#-Spezifikation zu *gl.Color_3f()* verändert. Diese, wenn auch geringe Veränderung, soll den erwarteten Datentyp eindeutig vom eigentlichen Befehl hervorheben. Ein anderes Beispiel wäre der Funktionsaufruf zum Verschieben von 3D-Objekten. Der originale Befehl *glTranslated()* setzt sich aus dem Präfix „gl“, der Funktion „Translate“ und dem Datentyp „d“ für d*ouble* zusammen. Hier wird deutlich, dass eine Trennung durchaus einen positiven Effekt hat. Bei dem neuen Methodenaufruf *gl.Translate_d()* steht das „d“ zweifelsfrei für d*ouble*.

Ein weiteres Beispiel für eine Anpassung an C# ist die Einführung der *Color* Klasse bei OpenGL-Befehlen. Natürlich kennt die OpenGL-Maschine keine komplexen Datentypen, die im .Net Framework ihren Ursprung haben. Aus dem Abschnitt „Einführung in OpenGL“ ist bekannt, dass für eine Farbzuweisung drei Werte zwischen Null und Eins erforderlich sind. Von diesem Standpunkt aus gesehen, müsste ein Farbobjekt vom Typ *Color* immer erst in seine drei Farbbestandteile zerlegt und umgerechnet werden, bevor es an die Funktion *Color3f()* übergeben werden kann. Um dem 3D-Grafik Programmierer diesen Aufwand abzunehmen, wurde die Erweiterung *Color_c()* zur *OpenGL* Klasse hinzugefügt. Das „c“ steht selbstverständlich für *Color*, da diese Methode ein *Color* Objekt entgegen nehmen kann. Diesem werden dann intern die vier *float* Werte für Rot, Grün, Blau und den Alphakanal entnommen und wie gewohnt an die *glColor4f()* Funktion weiter

gegeben. Alle Funktionen besitzen nun diese Erweiterung, die zuvor eine Farbe nur als RGB oder RGBA[18] Wertegruppe aufnehmen konnten.

4.5.3 Fazit

Die Visualisierungsbibliothek erfüllt eine wesentliche Aufgabe. Sie soll die Funktionalitäten von OpenGL so exakt wie möglich für die Hochsprache C# zur Verfügung stellen. Darüber hinaus wurde das Ziel erreicht, dass die Anwendung der 3D-Grafik Programmierung auch für Einsteiger interessant ist. Das bedeutet, dass die Initialisierung automatisiert wurde und ein, an das .Net Framework angepasstes, Steuerelement zur Nutzung bereit steht. Die Anwendung der C# spezifischen OpenGL-Befehle funktioniert analog zum normalen OpenGL-Gebrauch. Daher empfiehlt sich an dieser Stelle auch weiterführende Literatur, beispielsweise [ARB 08]. Im folgenden Kapitel soll nachgewiesen werden, dass sich die Eigenentwicklung für die dreidimensionale Simulation von Sensornetzen eignet.

4.6 Anwendung der Visualisierungsbibliothek

Die im voran gegangenen Abschnitt beschriebene *SharpOGL_AS.dll* soll gemäß der Aufgabenstellung in einer Anwendung implementiert und genutzt werden. Es bestünde an dieser Stelle die Möglichkeit, den Einsatz im Simulator/Demonstrator zu zeigen, da die selbst entwickelte Visualisierungsbibliothek vordergründig für diese Software erstellt wurde. Um die Abläufe etwas zu vereinfachen, soll nun jedoch eine Testanwendung herangezogen werden, die ohnehin während der Entwicklung der DLL entstand. Die Funktionalitäten der Testanwendung wurden soweit wie möglich für den Einsatz innerhalb des Simulators/ Demonstrators vorbereitet. Dabei entstand eine interaktive 3D-Umgebung, in der sich der Nutzer frei bewegen kann und Objekte sowohl selektieren, als auch

[18] RGBA steht für die drei Farbkanäle Rot, Grün, Blau und den Alphakanal, also die Transparenz der Farbe

verschieben kann. Die in den folgenden Abschnitten beschriebenen Methoden stammen stets aus der *SharpOGL_AS.dll,* mit der Anmerkung, dass bei Verwendung des *OpenGL* Objektes zu jeder Zeit der empfohlene Name „gl“ vergeben wurde. Dieses Kapitel soll keine Anleitung zum Erlernen von OpenGL darstellen. Daher wird nur auf die Literatur von Dave Shreiner "OpenGL Programming Guide: The Official Guide to Learning OpenGL" [ARB 08] verwiesen.

Einrichten des SceneGraphs

Nach dem Anlegen einer neuen Projektmappe muss man die Bibliothek einbinden. Die Abbildung 4-6 zeigt *Visual Studio*, nach dem Hinzufügen der DLL. Die rot markierten Bereiche zeigen auf der rechten Seite den neuen Verweis im Projektmappen-Explorer, sowie auf der linken Seite in der Toolbox das neue Steuerelement *SceneGraph*. Darüber hinaus ist das Objekt *sceneGraph1* bereits auf der Oberfläche der *Form1* zu sehen.

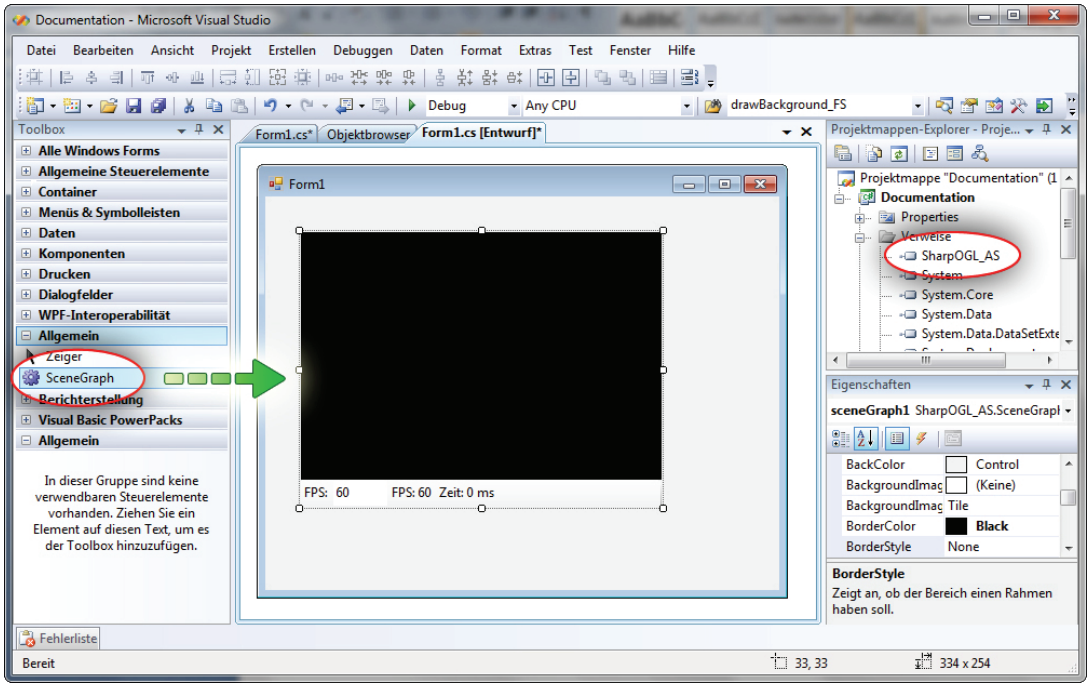


Abbildung 4-6 SceneGraph anlegen

Damit nun der *SceneGraph* als 3D-Ausgabemedium aktiviert wird, muss dessen wichtigstes Event, *OpenGL_Paint* angebunden werden. Die Abbildung 4-7 zeigt einen Teil der Liste der verfügbaren Events, mit der bereits hinzugefügten Eventmethode *sceneGraph1_OpenGL_Paint()*. Alle Ereignisse mit dem Präfix "OpenGL" sind für die 3D-Grafok Programmierung von Relevanz und wurden speziell für den *SceneGraph* entwickelt.

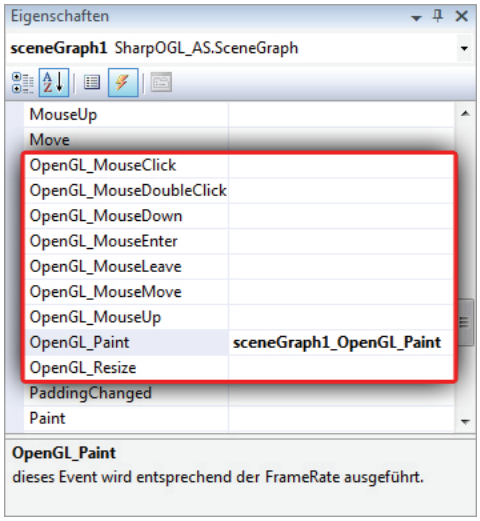


Abbildung 4-7 SceneGraph Ereignisse

Nun beginnt die eigentliche Programmiertätigkeit. Da die Visualisierungs-bibliothek objektorientiert aufgebaut ist, benötigt man ein *OpenGL* Objekt, was durch die 3D-Bibliothek mitgeliefert wird. Der Quellcode 4-2 zeigt dieses als global deklariertes Objekt *gl*. Diese Instanz stellt über ihre Methoden und Funktionen eine direkte Verbindung zum nativen OpenGL her. Der *SceneGraph* beginnt seinen Renderzyklus, indem man die Methode *startRendering()* ausführt. Ab diesem Zeitpunkt wird das Paint Event so oft in der Sekunde ausgeführt, wie es die *FrameRate* des *SceneGraphs* vorgibt. Die *startRendering()* Methode liefert unterdessen ein fertig initialisiertes *OpenGL* Objekt zurück, was an den *SceneGraphen* angekoppelt ist. Dieses Rückgabeobjekt muss als die globale

Instanz *gl* abgespeichert werden. In der *Load* Methode der *Form1* wurde dieser Schritt durchgeführt.

```
// Namensraum der DLL hinzufügen
using SharpOGL_AS;
using SharpOGL_AS.Definitions;

namespace Testsoftware
{
    public partial class Form1 : Form
    {
        // OpenGL Objekt "gl" um Renderbefehle auszuführen
        OpenGL gl;

        // Konstruktor der Form1
        public Form1()
        {
            InitializeComponent();
        }

        // Load Ereignis der Form1
        private void Form1_Load(object sender, EventArgs e)
        {
            // Rendering-Zyklus starten und gl Objekt global speichern
            gl = sceneGraph1.startRendering();
        }

        // Rendering-Zyklus
        private void sceneGraph1_OpenGL_Paint(object sender, PaintEventArgs e)
        {
            // Farb- und Tiefenpuffer leeren
            gl.Clear(GL_DEF.GL_COLOR_BUFFER_BIT | GL_DEF.GL_DEPTH_BUFFER_BIT);

            // Hintergrund grün zeichnen
            gl.ClearColor(Color.Green);
        }
    }
}
```

Quellcode 4-2 OpenGL Initialisierung

Nachdem das *Load*-Event der *Form1* abgearbeitet wurde, kann das *gl* Objekt für dreidimensionale Operationen eingesetzt werden. Durch die Bezeichnung lassen sich 3D-Algorithmen, die mit der C#-Spezifikation erstellt worden sind, nahezu genau so lesen, wie originale OpenGL-Anwendungen. Einziger Unterschied ist der Punkt, der zwischen *gl* und der aufgerufenen Funktion steht.

Die Abbildung 4-8 zeigt die Ausgabe, die durch den Quellcode 4-2 entstanden ist. Das gesamte Fenster ist mit dem *SceneGraphen* gefüllt worden. Die Statusleiste am unteren Rand ist ebenfalls ein Bestandteil des Steuerelementes. Diese Leiste

kann über die Eigenschaft *ShowStatusStrip* ein- beziehungsweise ausgeblendet werden. Mit Hilfe der Statusleiste kann man selbst während der Laufzeit einstellen, wie viele Bilder pro Sekunde maximal gerendert werden sollen. Neben diesem Eingabefeld sieht man noch die tatsächlich erreichten Frames pro Sekunde und die Zeitdauer, die der PC benötigte, um ein Bild zu berechnen.

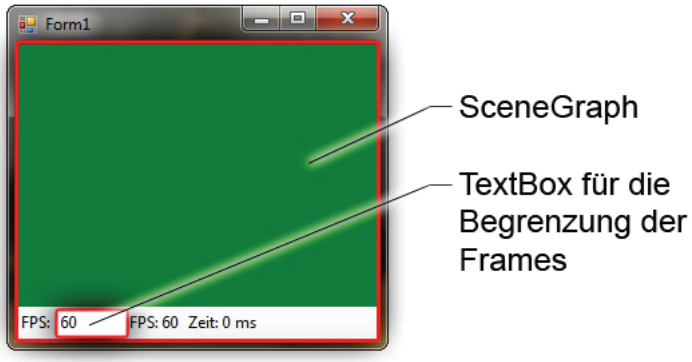


Abbildung 4-8 erste OpenGL Ausgabe

Nachdem nun bekannt ist, was die wichtigsten Eigenschaften des *SceneGraphen* sind, soll beschrieben werden, wie ein virtuelles Netzwerk dargestellt wird. Es soll eine Steuerung implementiert werden, wodurch sich der Nutzer frei im virtuellen Raum bewegen kann.

Die Kamera

Man muss an dieser Stelle stets berücksichtigen, dass es unter OpenGL kein tatsächliches Kameraobjekt gibt. Die Vorstellung einer Kamera soll nur helfen, sich in der dreidimensionalen Welt besser zurecht zu finden. Darüber hinaus, wurden einige Funktionen und Parameter bei OpenGL so benannt, als gäbe es eine Kamera, die man konfigurieren könnte. Für die Beschreibung der folgenden Thematik, soll ebenfalls der Begriff "Kamera" genutzt werden, um den Sachverhalt leichter begreifbar zu machen.

Aus dem Kapitel 2.4 "Grundbegriffe der 3D-Grafik Programmierung" ist bekannt, dass für gewöhnlich alle positiven Werte der Z-Achse hinter dem Blickfeld der Kamera liegen. Für die Positionierung von Netzwerkknoten, wäre es ein viel zu großer Aufwand, bei jeder Platzierung eines Knotens, ein negatives Vorzeichen beim Z-Wert anzugeben. Darüber hinaus entstünde eine Fehlerquelle, die vermieden werden sollte.

Die Lösung ist die *ModelView* Matrix. Man invertiert die Z-Achse, bevor man beginnt, die 3D-Umgebung und die Netzwerkteilnehmer zu rendern. Damit liegen alle Körper, die einen positiven Z-Wert besitzen, sofort im sichtbaren Bereich. Über den Skalierungsbefehl *gl.Scale_3f(1, 1, -1)* wird dieser Schritt erreicht. Die Skalierungsmethode erhält drei Übergabeparameter, die für die Achsen X, Y und Z stehen. Ein Wert kleiner 1 staucht die betreffende Achse, wogegen sie gestreckt wird, wenn man einen Wert größer 1 einsetzt. Bei -1 wird die Achse folglich gespiegelt und die Größenverhältnisse bleiben gleich.

Durch das Verhalten der *ModelView* Matrix lassen sich, bei geringer Änderung des Quellcodes, enorme Auswirkungen beim Rendering Ergebnis erreichen. So macht es beispielsweise einen großen Unterschied, ob man erst ein Objekt dreht und dann verschiebt, oder umgekehrt. Die Abbildung 4-9 zeigt die unterschiedlichen Ergebnisse, bei der Positionierung des grauen Quaders, wenn man die Reihenfolge der Programmierbefehle gl.Rotete_f() und gl.Translate_f() vertauscht.

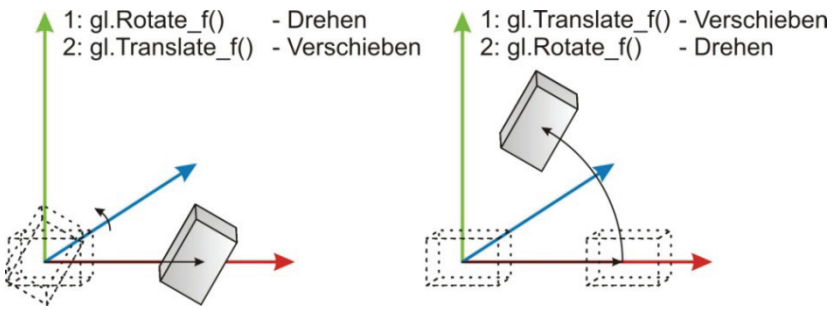


Abbildung 4-9 *ModelView* Matrix

Eine der Grundsatzdiskussionen bei OpenGL folgt dem Problem: „Dreht sich das Universum um die Kamera, oder dreht sich die Kamera um das Universum?“ Diese Frage lässt sich bei genauerer Untersuchung leicht beantworten. Verwendet man ausschließlich die Befehle *gl.Translate_f()*, *gl.Rotate_f()* und *gl.Scale_f()* zum Manipulieren der *ModelView* Matrix, dann steht die Kamera fest. Mit diesen drei Funktionen werden alle Objekte verschoben, gedreht oder skaliert, aber die Kamera befindet sich immer am Koordinatenursprung.

Die *Utility Library* bietet noch eine Funktion zum Manipulieren der Kamera. Mit *gl.glu.LookAt()* lassen sich die Position, ein Referenzpunkt und die Ausrichtung der Kamera im virtuellen Universum definieren. Für den Einsatz in einer dreidimensionalen Umgebung, in der sich der Anwender frei bewegen soll, ist diese Methode jedoch von Nachteil. Dies macht ein Vergleich beider Techniken deutlich.

Entwicklung der Kamerasteuerung

Ziel ist es, für die Testanwendung eine Steuerung zu entwickeln, mit der man sich in der Ego-Perspektive, auch „FirstPerson“ genannt, durch die Simulationsumgebung frei bewegen kann. Der Quellcode 4-3 zeigt die Umsetzung, wobei der Programmcode von allen Befehlen bereinigt wurde, die in keinem direkten Zusammenhang zur Ausgabe stehen. Damit man über die Maus und Tastatur mit der Software interagieren kann, muss man die Ereignisse für das Mausklicken und -Bewegen, sowie für das Drücken der Tasten, zur Anwendung hinzufügen. Die Winkel „xAngle“ und „yAngle“ aus dem Quellcode 4-3, stehen für die Neigung und Drehung der Kamera. Diese lassen sich über die Position der Maus, bezogen auf die Bildmitte, berechnen. Mit der Tastatur, beispielsweise mit den Pfeiltasten, lässt sich die Kamera vor, zurück, nach links und rechts bewegen. Die Variablen „PosX“, „PosY“ und „PosZ“ ergeben sich aus einem aufsummierten Betrag, der für

das Betätigen der Pfeiltasten festgelegt wurde. Setzt man nun diese Winkel- und Positionswerte in die Methoden gl.Rotate_f() und gl.Translate_f() ein, so hat man in wenigen Zeilen die Steuerung für eine „FirstPerson Kamerasteuerung“ realisiert. An dieser Stelle sei nochmals auf die irreführende Formulierung der Kamerabewegung hingewiesen. Tatsächlich dreht und bewegt sich alles, nur nicht die Kamera.

```
// der zyklische Rendervorgang
private void renderScene()
{
    // --- Kameraposition einstellen --- //

    // die Z Achse wird als erstes gespiegelt
    gl.Scale_f(1, 1, -1);

    // die Rotation der X und Y Achse auf den Matrix-Stapel legen
    gl.Rotate_f(xAngle, 1, 0, 0);
    gl.Rotate_f(yAngle, 0, 1, 0);

    // die gesamte Szene positionieren
    gl.Translate_f(PosX, PosY, PosZ);

    // --- jetzt kann die restliche Szene gerendert werden --- //
}
```

Quellcode 4-3 Steuerung Perspektive

Wie im Quellcode 4-3 ersichtlich, werden als erstes die X- und die Y-Achse, um den Rotationswinkel gedreht. Die drei Werte 1,0,0 beziehungsweise 0,1,0 stehen hierbei für die Achsenausrichtung in positiver X- und Y- Richtung. Im Anschluss daran wird die *ModelView* Matrix mit der Positionsverschiebung multipliziert. Nach dieser Konfiguration können alle Zeichenobjekte gerendert werden, sodass man den Eindruck einer natürlichen Bewegung in der virtuellen Welt erhält.

Die zweite mögliche Technik, wäre das "echte" Bewegen der Kamera. Die Methode *gl.glu.LookAt()* würde alle Anweisungen in einem Schritt erledigen. Dieser Befehl benötigt allerdings neun Parameter. Die Tabelle 4-2 beschreibt diese.

eyeX, eyeY, eyeZ	Position der Kamera
centerX, centerY, centerZ	Position des Referenzpunktes, auf den die Kamera fokussiert ist
upX, upY, upZ	Ein Richtungsvektor, der angibt, welche Seite der Kamera nach oben zeigt. Normalfall: 0,1,0

Tabelle 4-2 Übergabeparameter von gl.glu.LookAt()

Das größte Problem stellt der benötigte Referenzpunkt dar, auf den die Kamera zwangsläufig fest fixiert wird. Möchte man sich in der 3D-Umgebung bewegen, dann erfolgt automatisch auch eine Drehung, wenn man die Kameraposition verschiebt, ohne den Referenzpunkt mit zu bewegen. Dieser Blickpunkt lässt sich nicht über die Interaktion mit der Maus, oder der Tastatur errechnen. Unbekannte Variablen, wie beispielsweise Bewegungen des Referenzpunktes in Abhängigkeit der Entfernung zur Kamera, machen eine harmonische und natürliche Bewegung ungleich schwerer, als bei der ersten Möglichkeit. Unterschiede in der Performanz sind ebenfalls kein Argument für diesen Ansatz, da intern die *ModelView* Matrix auf die gleiche Weise beeinflusst wird, wie durch die Methoden *gl.Rotate_f()* und *gl.Translate_f()*. Der *gl.glu.LookAt()* Befehl übergibt lediglich in einem Schritt eine größere Anzahl an Parametern und wendet diese auf den Matrixstapel an. Aus diesen Gründen wurde auf den Einsatz dieser Variante verzichtet.

virtuelles Netzwerk

Die Testanwendung wurde im Hinblick auf die Simulation von Netzwerken gestaltet. Es wurde unter anderem eine Netzwerkattrappe entwickelt, wodurch große Teile des Quellcodes für den Simulator/Demonstrator wieder verwendet werden können. Dieses virtuelle Netzwerk besteht im Grunde nur aus einer Reihe von zufällig errechneten Positionen im Raum. An jede Koordinate, die für den Standpunkt eines Sensorknotens steht, soll eine blaue Kugel gezeichnet werden.

Für den geplanten Einsatz in der Simulationssoftware müssen diese Zufallskoordinaten nur noch durch die Werte ersetzt werden, die von dem Simulationsprozess vorgegeben werden.

Die Darstellung der Kugeln erfolgt über die GLU. Die *OpenGLUtilityLibrary* bietet unter anderem Zeichenbefehle zum Rendern von Kugeln, Scheiben, Kegeln oder Zylindern. Mit der Funktion *gl.glu.Sphere()* wird eine Kugel um den Koordinatenursprung herum gezeichnet. Der Quellcode 4-4 zeigt, welche Parameter an die Funktion übergeben werden müssen. Das *quadric* Objekt wird ebenfalls über die *GLU* erzeugt und dient als Hüllobjekt für die Primitiven, aus denen die Kugel besteht. Der *radius* bestimmt die Größe der Kugel. Die *Integer* Werte *slices* und *stacks* legen die Anzahl der Unterteilungen fest, aus denen die Kugel schließlich besteht. Die Abbildung 4-10 bildet das grafisch überarbeitete Ergebnis ab, welches durch den Quellcode 4-4 entstehen würde

```
private void renderSphere()
{
    IntPtr quadric = gl.glu.NewQuadric();
    double radius = 1;
    int slices = 24;
    int stacks = 12;

    // jetzt wird die Kugel gerendert
    gl.glu.Sphere(quadric, radius, slices, stacks);
}
```

Quellcode 4-4 SharpOGL Kugel Rendering

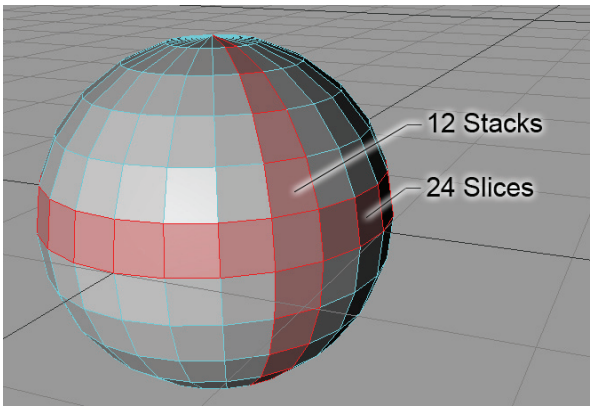


Abbildung 4-10 Kugelobjekt

Bevor jedoch ein schematischer Sensorknoten gerendert werden kann, muss die Kugel noch an die dazu gehörigen Koordinaten verschoben werden. Dies geschieht wieder mittels der *ModelView* Matrix, die nach jeder Kugel wieder zurück gesetzt werden muss.

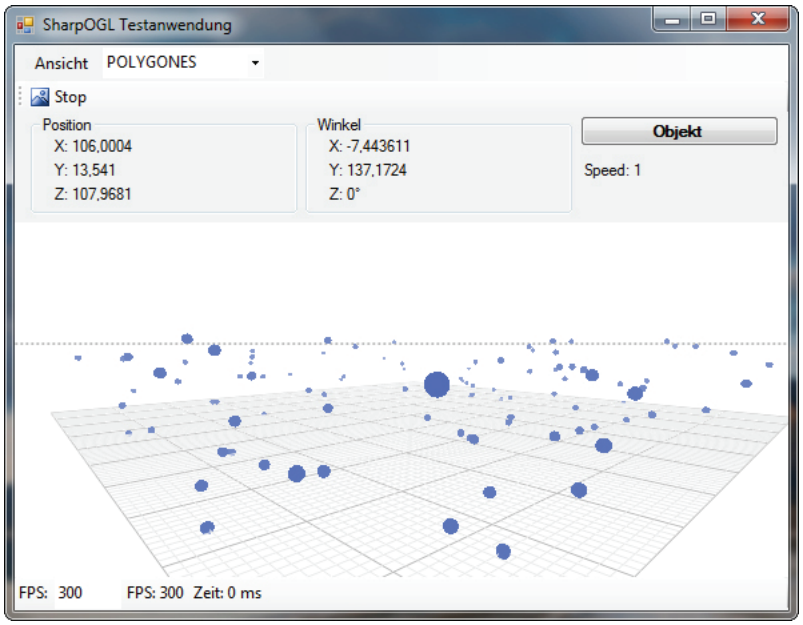


Abbildung 4-11 3D Sensorknoten

Die Abbildung 4-11 zeigt 100 zufällig platzierte Netzwerkknoten auf einer Fläche von 100 x 100 x 30 Längeneinheiten. Damit man sich besser im dreidimensionalen Raum orientieren kann, wurde in die Simulationsumgebung noch eine Horizontlinie und Gitternetz bei Y gleich Null implementiert. Oberhalb des *SceneGraphs* sind noch einige Hilfsausgaben hinzu gekommen, wie zum Beispiel die Anzeige, an welcher Position und in welchem Winkel sich die Kamera momentan befindet.

Beleuchtung

Wie man auf der Abbildung 4-11 sehen kann, erwecken die Kugeln noch keinen räumlichen Eindruck. Dies geschieht durch das Hinzufügen einer Lichtquelle. Für

die Testsoftware entschied man sich dafür, eine kugelförmige Lichtquelle zu erzeugen, die sich immer an der Position der Kamera befindet. Aus dem Grundlagenkapitel 2.4 ist bekannt, was es für Lichtarten unter OpenGL gibt. Der Quellcode 4-5 zeigt die Methode *setLightning()*, welche in der Testapplikation zum Einsatz kommt. Diese Prozedur muss nur einmal beim Start der Anwendung durchlaufen werden, um das Lichtobjekt zu konfigurieren. Dadurch bleiben die Eigenschaften der Lichtquelle, bis auf die Position, während der gesamten Laufzeit unverändert.

```
private void setLightning()
{
    // Position, Ambient, Diffuse und Specular von LIGHT0 einstellen
    gl.Light_fv(GL_DEF.GL_LIGHT0, GL_DEF.GL_POSITION,
                new float[] {0,0,0});

    gl.Light_fv(GL_DEF.GL_LIGHT0, GL_DEF.GL_AMBIENT,
                new float[] {0.5f,0.5f,0.5f,1.0f});

    gl.Light_fv(GL_DEF.GL_LIGHT0, GL_DEF.GL_DIFFUSE,
                new float[] { 0.5f, 0.5f, 0.5f, 1.0f });

    gl.Light_fv(GL_DEF.GL_LIGHT0, GL_DEF.GL_SPECULAR,
                new float[] { 0.77f, 0.77f, 0.77f, 1.0f });

    // lineare Abnahme des Lichtes einstellen
    gl.Light_f(GL_DEF.GL_LIGHT0, GL_DEF.GL_LINEAR_ATTENUATION, 0.005f);

    // Reflexionsverhalten der Lichtarten für Oberflächen einstellen
    gl.Material_fv(GL_DEF.GL_FRONT, GL_DEF.GL_AMBIENT,
                   new float[] { 0.23f, 0.23f, 0.23f, 1.0f });

    gl.Material_fv(GL_DEF.GL_FRONT, GL_DEF.GL_DIFFUSE,
                   new float[] { 0.28f, 0.28f, 0.28f, 1.0f });

    gl.Material_fv(GL_DEF.GL_FRONT, GL_DEF.GL_SPECULAR,
                   new float[] { 0.77f, 0.77f, 0.77f, 1.0f });

    // Beleuchtung wird aktiviert
    gl.Enable(GL_DEF.GL_LIGHTING);
    // Lichtquelle 1 von 8 wird aktiviert
    gl.Enable(GL_DEF.GL_LIGHT0);
    // Materialeigenschaften werden aktiviert
    gl.Enable(GL_DEF.GL_COLOR_MATERIAL);

}
```

Quellcode 4-5 Beleuchtung

Wie man an dem Beispielcode 4-5 sehen kann, gib es im Grunde nur einen Befehl für die Einstellung sämtlicher Beleuchtungsangelegenheiten. Die Funktionen *gl.Light_fv()* und *gl.Light_f()* gleichen sich, bis auf den Unterschied, dass bei der

Vektorvariante ein *float Array* statt einer einzelnen Gleitkommazahl erwartet wird. Der erste Übergabeparameter definiert, welches der acht OpenGL Lichtobjekte eingestellt werden soll. Die gewünschte Eigenschaft, die konfiguriert werden soll, wird über den zweiten Parameter festgelegt. Als letzter Parameter folgt schließlich ein *Array* vom Typ *float.* Es übergibt die Einstellungen, die angewendet werden sollen. Bei der Positionierung, die über *GL_POSITION* erreicht wird, handelt es sich bei dem *float* Array um die X-, Y- und Z-Koordinate, die das Lichtobjekt im virtuellen Raum einnehmen soll. Für die drei Lichtarten *GL_AMBIENT, GL_DIFFUSE* und *GL_SPECULAR* werden vier Gleitkommazahlen zwischen Null und Eins erwartet, die für die drei Farbkanäle Rot, Grün und Blau stehen. Der vierte *float* Wert bestimmt den Alphaanteil des Lichtes.

Das Erscheinungsbild von Materialoberflächen lässt sich ähnlich festlegen, wie bei Lichtquellen. Die Methode *gl.Material_fv()* kann, wie in Quellcode 4-5 dargestellt, die drei Lichtarten für Polygone definieren. Daneben gibt es aber noch weitere Einstellmöglichkeiten, wie beispielsweise Glanz oder Lichtemission von Oberflächen.

Damit nun in einer 3D-Anwendung diese Einstellungen tatsächlich ausgeführt werden, müssen diese noch in der Zustandsmaschine aktiviert werden. Mit *gl.Enable()* und der betreffenden Konstante wird dieser Schritt ausgeführt. Im Anschluss an die Einstellung der Szenenbeleuchtung wird die Hauptschleife des Rendervorgangs gestartet. In der Abbildung 4-12 ist das Testprogramm zu sehen, was um die *setLightning()* Methode erweitert wurde.

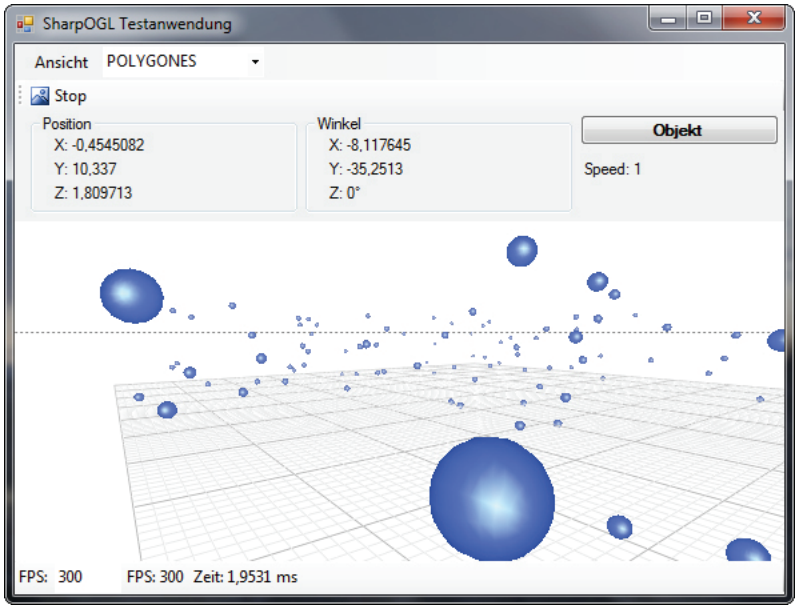


Abbildung 4-12 Ergebnis der Beleuchtung

Objektselektion

Eine virtuelle Umgebung, in der man sich frei bewegen kann, ist ohne großen Nutzen, wenn man mit ihr nicht in Kontakt treten kann. Die Nutzerinteraktion ist daher ein zentraler Schwerpunkt bei der Anwendung der *SharpOGL_AS.dll*. Für die Darstellung der Simulation von Netzwerken benötigt man eine Vielzahl von Objekten. Darunter zählen unter anderem ein Koordinatensystem, Netzwerkteilnehmer aller Art und die Umgebung an sich. In dieser 3D-Welt gibt es Bestandteile, die fest stehen sollen und andere, die vom Nutzer bewegt werden dürfen. Im Fall des Simulator/Demonstrator sind dies vordergründig die Netzwerkknoten. Sie müssen mit der Maus platziert und verschoben werden können. Dazu muss man beispielsweise wissen, um welches Objekt es sich handelt, welches sich momentan unter dem Mauszeiger befindet.

Der erste Schritt in diese Richtung ist die Möglichkeit der Objekterkennung. Für gewöhnlich kennt OpenGL nur Punkte, Linien und Polygone und weiß daher nicht, welche Primitive zu einem zusammenhängenden Körper gehören und welche

nicht. Um nun komplexere Objekte erkenntlich zu machen, kann man an alle Primitive, die auf dem Monitor angezeigt werden, Namen vergeben. Der Begriff „Name“ ist an dieser Stelle recht unpräzise, da es sich eigentlich um vorzeichenlose *Integer* Werte handelt. Wie es bei der Zustandsmaschine üblich ist, haben alle Primitive denselben Namen, bis ein Anderer zugewiesen wird. Es empfiehlt sich also, vor jedem Algorithmus, zum Zeichnen eines abgegrenzten Körpers, einen neuen Wert auf den Namensstapel zu legen. Im Anschluss an diese Maßnahme, besitzt jedes eigenständige Objekt eine Art Identifikationsnummer.

Für diese Aufgabe werden zwei OpenGL-Befehle zur Verfügung gestellt. Die erste Funktion *gl.InitNames()* dient zum Initialisieren des Namenstapels. Diese Methode muss zwingend ausgeführt werden, da ohne Sie keine Namensvergabe möglich ist. Später können beliebig viele Namen auf den Stapel gelegt werden. Der Befehl zum Hinzufügen eines neuen Namens lautet *gl.PushName().* Die Funktion erwartet als Übergabeparameter eine fortlaufende positive Ganzzahl. Es ist allerdings zwingend erforderlich, dass der erste Eintrag in der Namensliste eine Null ist, da sonst ein Laufzeitfehler durch OpenGL ausgelöst wird.

Damit ermittelt werden kann wo, beziehungsweise welche Maustaste gedrückt wurde, werden die Maus Ereignisse genutzt, die bereits zur Steuerung der Kamera dienten. Das Event *OpenGL_MouseDown* fängt ab, wann und wo eine Maustaste auf der 3D-Oberfläche gedrückt wurde. Wird nun dieses Ereignis ausgelöst, muss ein Algorithmus ausgeführt werden, der in der Lage ist, den Namen des 3D-Objektes zu bestimmen, falls auf einen Körper geklickt wurde. Man muss bedenken, dass man sich in einer virtuellen Umgebung befindet, worin eine Vielzahl von Objekten existiert, die aus allen Richtungen betrachtet werden können. Es ist also keineswegs trivial zu ermitteln, welcher Körper sich gerade am dichtesten vor der Kamera befindet und von welcher Position aus dieser zu sehen ist.

Da dieses Problem in vielen 3D-Anwendungen eine Rolle spielt, hat sich ein bestimmter Algorithmus, zur Detektion von Objekten bei einem Mausklick durchgesetzt. Er ist in nahezu unveränderter Form in einigen Internet Foren und Tutorials[19] zu finden. Diese üblicherweise verwendete Prozedur wurde für den objektorientierten Gebrauch mit der *SharpOGL_AS.dll* angepasst. Der in die Testanwendung implementierte Algorithmus, kann in den Anlagen unter Quellcode A-1 mit einer Quellenangabe nachgelesen werden.

Vereinfachend kann man sich vorstellen, dass bei einem Mausklick, an der Position des Zeigers, ein Loch durch die gesamte Szene gebrannt wird. Alle Namen der getroffenen Objekte, werden bis zu einem maximalen Z-Wert, also in die Monitorebene hinein, in einem *uint Array* mit der dazu gehörigen Tiefenposition gespeichert. Aus dieser Liste von Objektnamen, kann nun der Name gezogen werden, welcher sich an der niedrigsten Z-Position, bezogen auf die Kamera, befindet. Die Abbildung 4-12 soll dies veranschaulichen.

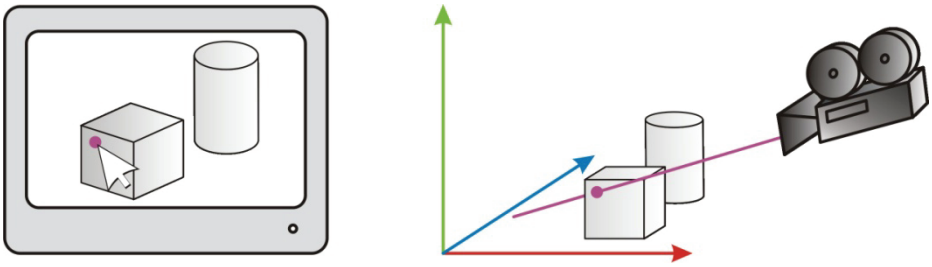


Abbildung 4-13 Objekteselektion Schema

Der linke Bereich entspricht der Monitorausgabe, die den Betrachtungswinkel zeigt, den man während der Laufzeit der Software sehen würde. Der Violett dargestellte Strahl auf der rechten Seite symbolisiert den Bereich, in dem Objekte aufgrund des Mausklicks detektiert werden. Wie man unschwer erkennen kann, ist es vom Betrachtungsstandpunkt abhängig, welche Körper bei einer Objekt-

[19] eine Anleitung, in der Schritt für Schritt eine Problematik beschrieben und gelöst wird

selektion ausgewählt werden, und welche nicht. Entlang der Z-Achse der Kamera wird die Detektion durchgeführt. Es gibt also einen Unterschied zwischen der Achsenausrichtung der gesamten Szene und der Richtung der Achsen, die sich auf den Betrachtungswinkel beziehen. Wenn auf einen Körper geklickt wurde, liefert der Algorithmus A-1 den Namen, das heißt die Nummer des gewünschten Objektes zurück. Der graue Würfel aus der Abbildung 4-13 könnte nun verschoben werden.

Im Hinblick auf die Simulationsanwendung ist es notwendig, die Netzwerkknoten dynamisch, also während der Laufzeit bewegt werden können. Aus diesem Grund wurde eine Hüllklasse namens *Node3D* entwickelt, welche die bereits beschriebenen blauen Kugeln abbildet. Einer Instanz dieser *Node3D* Klasse, kann man Raumkoordinaten zuweisen, eine ID, die den Namen repräsentiert und eine Farbe, mit der man beispielsweise einen Energiezustand eines Sensorknotens darstellen könnte. Des Weiteren verfügt die Klasse über einen Rendering-Befehl, der dann in der Hauptschleife des 3D-Programms ausgeführt wird. Dank der *Node3D* Klasse, können die Instanzen ohne großen Aufwand selektiert und verschoben werden, weil die Positionsdaten zur Laufzeit verändert werden können.

Wie nun im Detail die Körper während der Laufzeit bewegt werden, ist anwendungsspezifisch und kann auf unterschiedlichste Art und Weise umgesetzt werden. Für die Testapplikation wurden die Tasten "1", "2" und "3" als Steuertasten eingerichtet, die beim Betätigen, eine Verschiebung des selektierten Objektes auf der X-, Y- oder Z-Achse frei schalten. Wenn zu einer der gedrückten Tasten zusätzlich noch die linke Maustaste gedrückt wird, dann bewegt sich das Objekt entlang der gewünschten Achse, je nach Mausbewegung.

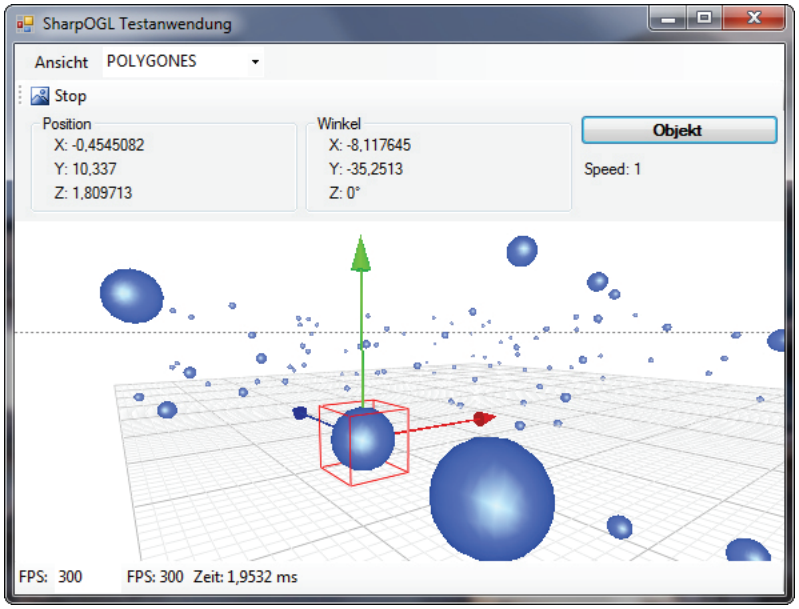


Abbildung 4-14 Objektselektion Ausgabe

In der Abbildung 4-14 ist der Zeitpunkt zu sehen, als gerade eine blaue Kugel aus der Masse selektiert wurde und vor die Kamera gezogen wurde. Die selektierten Körper werden durch einen roten Rahmen gekennzeichnet. Die Pfeile zeigen entlang der X-, Y- und Z-Achse, die farblich unterschiedlich dargestellt werden. Rot steht für X, grün für Y und die Z-Achse wird blau gezeichnet. Wenn man nun, die bereits erwähnten Steuertasten drückt, dann ändert sich die entsprechende Pfeilfarbe zu gelb, sodass der Nutzer weiß, dass er nun den Körper entlang dieser Achse verschieben kann.

4.7 Fazit

Die Entwicklung, Realisierung und Implementierung der Visualisierungsbibliothek hat den größten Teil der Bearbeitungszeit dieser Diplomarbeit beansprucht. Während dieser Zeit entstand eine einsatzfähige Bibliothek, mit der man unter C# auf die Funktionalitäten von OpenGL zugreifen kann. Der Anforderung an die DLL, dreidimensionale Netzwerke darstellen zu können, konnte entsprochen werden, wie die Testanwendung zeigt. Es wurde unter anderem auf eine objektorientierte

Struktur Wert gelegt, was für die Anwendung der Wrapper-DLL einige Vorteile mit sich bringt. Die *SharpOGL_AS.dll* ist an die Arbeit mit *Visual Studio* angepasst und besitzt einen *SceneGraphen*, der die 3D-Grafik Programmierung komfortabler gestaltet.

Die Visualisierungsbibliothek konnte mit kostenfreien Lösungskonzepten aus dem Internet verglichen werden. Daraus wurde ersichtlich, dass die Eigenentwicklung am besten an die Simulationssoftware angepasst ist. Hinzu kommt, dass allein durch den Aufbau eigener Kompetenzen, eine Weiterentwicklung der dreidimensionalen Darstellung gewährleistet werden kann.

Mit der Testsoftware, die parallel zur Entwicklung der *SharpOGL_AS.dll* entstand, konnten zahlreiche Funktionalitäten realisiert werden, die auch im Simulator/ Demonstrator ihren Einsatz finden werden. Dazu zählt in erster Linie die interaktive Steuerung in einer 3D-Umgebung, die Interaktion mit virtuellen Objekten und die Darstellung einer Simulationsumgebung für schematisch dargestellte Netzwerke. Auf diesem Grundwissen aufbauend, kann nun die Weiterentwicklung und Optimierung der Visualisierungsbibliothek folgen.

5 Zusammenfassung und Ausblick

5.1 Erreichte Ergebnisse

Im Verlauf dieser Arbeit wurden zwei bedeutende Ziele erreicht. Zum einen wurde eine modular aufgebaute Softwareoberfläche für eine Simulationssoftware entwickelt und realisiert. Zum anderen wurde eine Visualisierungsbibliothek konzipiert, erstellt und implementiert, mit deren Hilfe man direkt auf die Grafikschnittstelle OpenGL, aus dem .Net Framework heraus, zugreifen kann.

Simulator/Demonstrator

Es wurden zahlreiche Oberflächeninhalte realisiert, mit denen die Simulation drahtloser Netzwerke angelegt, dargestellt und ausgewertet werden kann. Die Abbildung 5-1 zeigt den pausierten Simulationsablauf mit einer Oberflächen-konfiguration, die drei aktive Fenster zeigt. Oberhalb der "2D Umgebung" ist die Steuerung des Simulationsprozesses zu sehen. Die Simulation, kann hier gestartet gestoppt, pausiert oder im Einzelschritt-Betrieb ausgeführt werden.

Das wichtigste Anzeigefenster ist die "2D Umgebung". Hier sieht man alle während der Simulation aktiven Netzwerkteilnehmer, als farbige Kreise dargestellt. Man erkennt die Positionen der Netzwerkknoten in der Simulations-umgebung und die momentan bestehenden Funkverbindungen, die durch farbige Linien dargestellt werden. Das Clustering des Adhoc-Netzwerkes ist somit zu jedem Zeitpunkt deutlich zu sehen. Es besteht außerdem die Möglichkeit, sich ToolTips[20] zu jedem Netzwerkteilnehmer anzeigen zu lassen.

[20] Zusatzinformationen, die mit durch eine Mausbewegung eingeblendet werden können

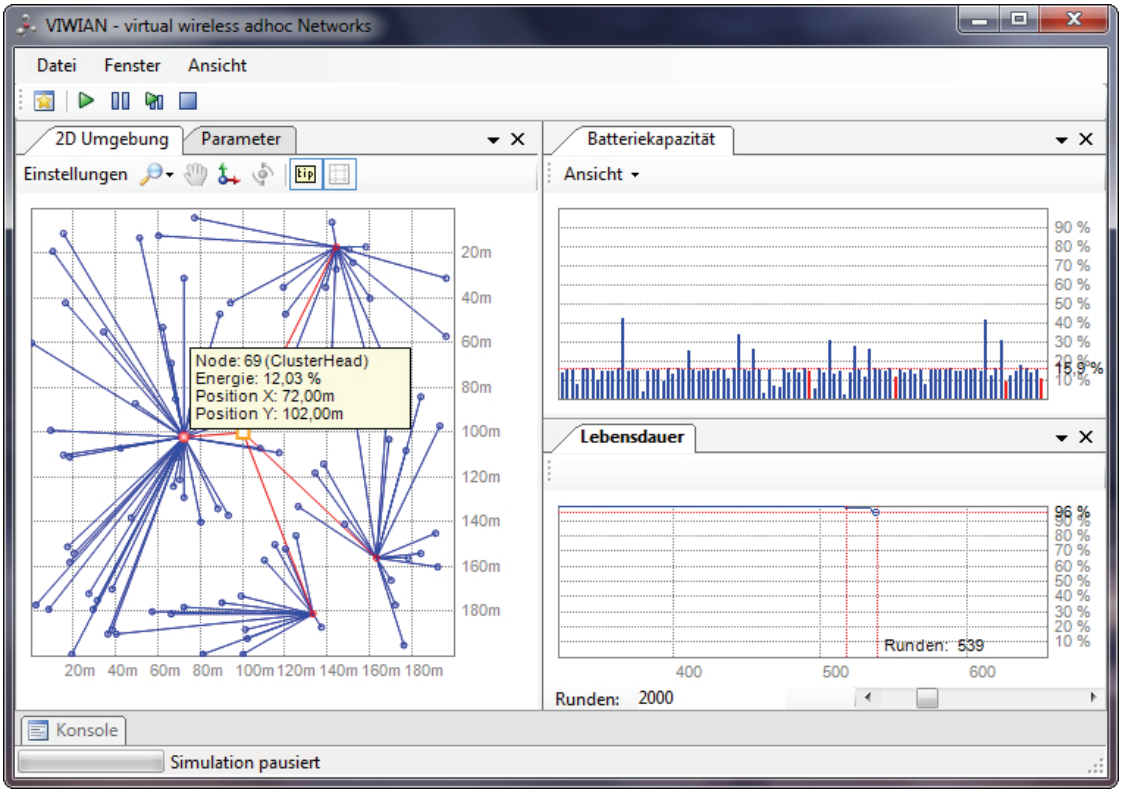


Abbildung 5-1 pausierter Simulationsprozess

Der Lebensdauergraph unten rechts zeigt, wie viele Knoten noch aktiv sind und gibt dieses Ergebnis in Prozent bei jedem Kommunikationszyklus aus. Ein weiteres interessantes Diagramm zeigt die Batteriekapazität eines jeden Knotens. Dieses Diagramm oben rechts kann als Balken- oder als Liniendiagramm angezeigt werden. Darüber hinaus gibt es den durchschnittlichen Energiewert aller Netzwerkteilnehmer aus. Alle Diagramme werden dynamisch berechnet und dargestellt, sodass eine Größenänderung oder eine Änderung der Seitenverhältnisse automatisch angepasst wird. Es wurden weiterhin Tabellen zur Auswertung der Simulation implementiert, um detaillierte Informationen einsehen zu können.

Neben den angezeigten Fenstern entstanden noch weitere Inhalte, unter anderem ein Einstellungsfenster, womit die Netzwerkparameter dynamisch verwaltet werden können. Die Abbildung 5-2 zeigt den Einstellungsdialog für die

Netzwerkprotokolle. Die Darstellung der Parameter erfolgt über eine *GridView* Tabelle. Diese wurde um ein selbst entwickeltes Steuerelement *GridEditor* (rot hervorgehoben) erweitert, wodurch die sensible Konfiguration einer verbesserten Fehlervalidierung unterzogen werden konnte.

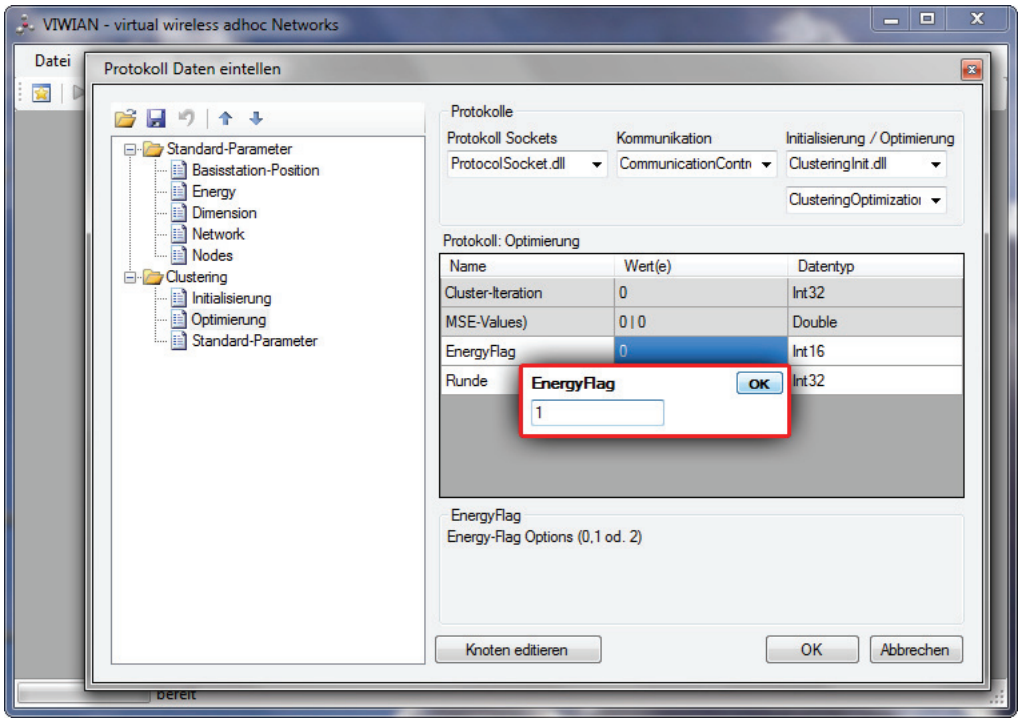


Abbildung 5-2 dynamisches Einstellungsfenster für das Simulationsprotokoll

Zusammenfassend lässt sich feststellen, dass die Anforderungen, die an die Anwendungsoberfläche gestellt wurden, erfüllt worden sind. Durch den modularen Aufbau der GUI ist diese auch in ferner Zukunft schnell und leicht erweiterbar. Der Grundaufbau der Programmoberfläche basiert auf einem *DockContent* Konzept, welches als Open Source Variante aus dem Internet zur Verfügung steht. Mit Hilfe der Bibliothek von Weifen Luo, wird es dem Nutzer ermöglicht, die Fenster der Computersoftware zu verwalten und während der Laufzeit, je nach Belieben, einzurichten. Dem wurde eine Speicher- und Ladefunktionalität für die Fenstereinstellung hinzugefügt. Des weiteren wurde der Simulator/Demonstrator von Beginn an als mehrsprachig funktionierende

Software konzipiert. Dazu wurde eine eigenständig entwickelte *Languages.dll* entworfen und angebunden. Mit dieser Sprachbibliothek können, unabhängig vom Softwareprojekt, neue Sprachen in die Simulationssoftware implementiert werden. Damit lässt sich die Softwareoberfläche ohne Programmierkenntnisse in neue Sprachen übersetzen.

Die Visualisierungsbibliothek

Die Visualisierungsbibliothek, mit der man auf die Grafikschnittstelle OpenGL zugegriffen werden kann, wurde als Wrapper-DLL designt. Das bedeutet, dass die nativen OpenGL-Funktionen in nahezu unveränderter Form in die Programmiersprache C# übersetzt wurden. Hinzu kommt, dass die geplante Entwicklung eines *SceneGraphen* realisiert wurde. Mit Hilfe dieses Steuerelementes, konnte dem Anspruch an eine komfortable und objekt-orientierte Anwendung der *SharpOGL_AS.dll* nachgekommen werden. Der *SceneGraph* führt bei seiner Instanziierung alle Initialisierungsvorgänge der OpenGL-Maschine durch, sodass sich der Programmierer einer 3D-Anwendung nicht mit dem Einrichten von OpenGL befassen muss. Der *SceneGraph* besitzt zusätzlich nützliche Eigenschaften und Events, die in der IDE von *Visual Studio* eingestellt werden können. Dazu zählt die *FrameRate*, also die maximal zu berechnenden Bilder pro Sekunde. Weitere Beispiele sind das Paint Event und die Ereignisse für Maus- und Tastaturinteraktionen.

Parallel zu der Arbeit an der Visualisierungsbibliothek, musste ein Testprogramm entwickelt werden, wodurch der Fortschritt der 3D-Bibliothek erst sichtbar wurde. Die Abbildung 5-3 zeigt zwei unterschiedliche Momentaufnahmen. In dem linken Fenster ist eine zufällige Anordnung schematischer Netzwerkknoten in einer Simulationsumgebung zu sehen. In dem rechten Fenster wurde eine Landschaft mit Hilfe einer Bitmap generiert und dargestellt.

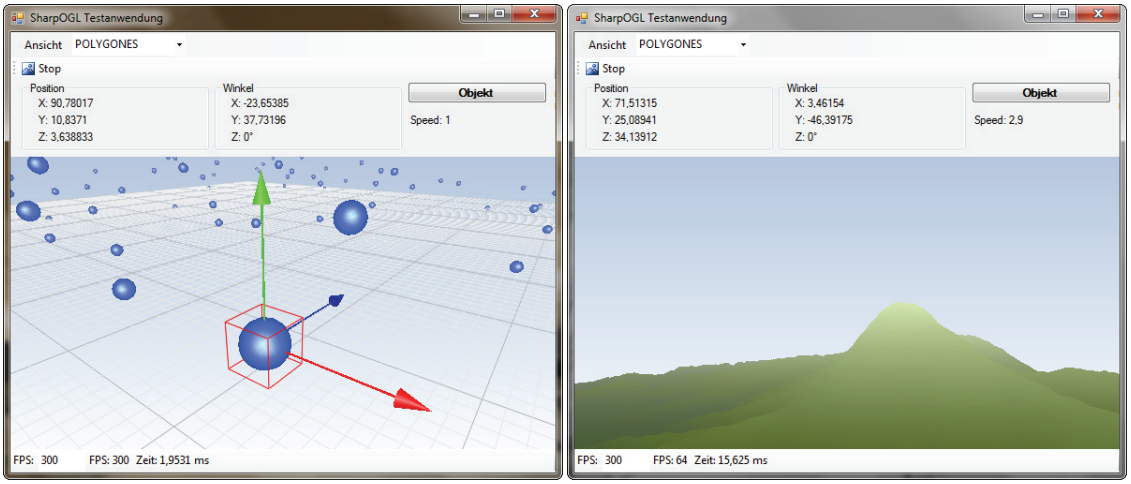


Abbildung 5-3 Testanwendung für SharpOGL_AS.dll

Die Testanwendung verfügt über eine Steuerung aus der Ego-Perspektive heraus. Das bedeutet, dass man sich mit den Pfeiltasten oder mit W, A, S und D und einer Mausbewegung frei bewegen kann.

5.2 Ausblick auf die Simulationssoftware

Die nächste Weiterentwicklung im Bezug auf den Simulator/Demonstrator muss die Implementierung der 3D-Visualisierungsbibliothek sein. Damit werden die zwei großen Teilaufgaben wieder zu einem Ergebnis zusammengeführt: die dreidimensionale Simulation und Visualisierung von drahtlosen Sensornetzwerken. Da die Testanwendung der 3D-Bibliothek bereits im Hinblick auf dieses Ziel entwickelt wurde, können nahezu alle Funktionalitäten aus diesem Beispielprogramm übernommen werden. Die Anbindung des Simulationsvorgangs kann analog von der 2D-Darstellung übernommen werden. Es ist geplant, dass noch weitere Oberflächeninhalte in die Simulationsanwendung integriert werden. Dazu zählt unter anderem das Vorhaben, mehrere Simulationsprozesse parallel ablaufen zu lassen. Für diese und weitere Aufgaben muss die GUI des Simulators/Demonstrators auch künftig weiter entwickelt werden.

5.3 Ausblick auf Erweiterungen der Visualisierungsbibliothek

Die Entwicklung einer Visualisierungsbibliothek für OpenGL-Funktionalitäten unter C# wurde erfolgreich begonnen. Mit diesem positiven Einstieg in die Materie wurde der Grundstein für eine langfristige Weiterentwicklung gelegt. Während der Bearbeitungszeit entstanden viele neue Ideen und Ansätze, die nun angesprochen werden sollen.

Wichtigste Aufgabe ist eine Verbesserung der Leistungsfähigkeit der Wrapper-DLL. Darunter zählt ein Ausbau der Funktionen, um nach Möglichkeit ein vollständiges Abbild der OpenGL-Grafikschnittstelle zu erhalten. Neben dem Bereitstellen der normalen Grundfunktionen wäre auch eine Komponente vorstellbar, die bereits komplexere Anweisungen erlaubt. Dazu könnte die Eigenentwicklung oder die Implementierung einer bestehenden Physik Engine[21] zählen. In Abbildung 5-4 wird solch eine physikalische Komponente eindrucksvoll veranschaulicht. Eine mit *Cinema4D* [C4D 09] modellierte Pyramide, die aus zahlreichen aufgestellten Quadern besteht, stürzt allein aufgrund der aktivierten Physik Engine zusammen. Dabei ist das korrekte Zusammenspiel von Schwerkraft, Kollisionen, kinetischer sowie potentieller Energie der Objekte zu berechnen.

Abbildung 5-4 Physik Engine von Cinema4D

[21] deutsch: Physikmaschine, berechnet physikalische Vorgänge von 2D und 3D Programmen

Die physikalischen Fähigkeiten einer Simulation spielt eine bedeutende Rolle, da erst durch diese ein möglichst realistisches Abbild der Wirklichkeit entstehen kann. Es könnten beliebige Bereiche aus der Physik, beispielsweise die Kinetik oder das Verhalten von Wellen und Teilchen, simuliert werden. Ein geplantes Ziel ist die virtuelle Abbildung des physikalischen Funkkanals eines drahtlosen Netzwerkes. Dieser Gedanke eröffnet ein neues und gleichermaßen umfassendes Tätigkeitsfeld. Dazu zählt die Visualisierung des Verhaltens von Funkwellen, Kollisionserkennung, Durchlässigkeit von Materialien und vieles mehr.

Ein weiterer Punkt bei der Leistungssteigerung der *SharpOGL_AS.dll* ist die Analyse der Performanz. Die Visualisierungsbibliothek basiert auf dem .Net Framework und es ist allgemein bekannt, dass leistungskritische Anwendungen ohne Verwendung des Frameworks schneller ablaufen. Ausführliche Messungen der Leistungsfähigkeit, beziehungsweise Maßnahmen zur Leistungsoptimierung sind während der Bearbeitungszeit kaum möglich gewesen. Es besteht die Möglichkeit, die Wrapper-DLL mit so genanntem "nicht verwalteten Programmcode" auszustatten. Allerdings müsste überprüft werden, in wie weit der Einsatz von unsicherem Code erforderlich ist und welche Auswirkungen das auf die Struktur der DLL hat.

Im Kapitel 4.4 "andere Lösungskonzepte" wurde die "NeHe Helix", eine Animation die nun als Referenz dienen soll, vorgestellt. Das gleiche Beispiel wurde zum Vergleich auch mit der *SharpOGL_AS.dll* programmiert. Dabei stellte sich heraus, dass der Animationszyklus der Helix ungefähr 25% langsamer abläuft, als mit der 3D-Bibliothek *CsGL*. Die Ursachen für diesen Leistungsunterschied müssen noch weiter untersucht werden. Es müsste also eingehender geprüft werden, in wie weit es Geschwindigkeitsprobleme bei der Darstellung von aufwändigeren Szenen gibt.

Es gibt noch weitere Ansätze für Weiterentwicklungen, die beispielsweise in einer Folgearbeit realisiert werden könnten. Dazu zählt die Implementierung eines Fehlermanagement-Systems, einer Funktionalität zum Laden von Modellen (zum Beispiel Häuser-Modelle oder vorgefertigte Simulationsumgebungen) oder die Eigenentwicklung eines Modell Editors. Damit soll zum Ausdruck gebracht werden, dass ein sehr großes Potential in der Weiterentwicklung dieser 3D-Visualisierungsbibliothek steckt.

Thesen der Arbeit

- Die Entwicklung drahtloser Sensornetze verlangt nach modernen Softwaresystemen, die qualitativ hochwertige Simulationen ermöglichen.
- Die Gebrauchstauglichkeit der Softwareoberfläche ist ein wesentlicher Bestandteil bei der Verwendung von wissenschaftlichen Simulationsanwendungen.
- Der modulare Aufbau einer Softwareoberfläche bietet dem Nutzer eine höhere Individualisierbarkeit der Anwendung und dem Entwickler eine übersichtlichere Programmstruktur.
- Die 3D-Grafikschnittstelle OpenGL ist für professionelle und wissenschaftliche Anwendungen sehr gut geeignet.
- Die dreidimensionale Darstellung von virtuellen Netzwerken ist eine deutliche Verbesserung für die detaillierte Anzeige und Auswertung der Simulationsabläufe.
- Derzeit existieren kaum geeignete Lösungskonzepte, um auch innerhalb des .Net Frameworks auf OpenGL zugreifen zu können.
- Das Ergebnis dieser Arbeit ist die einsatzbereite Anwendungsoberfläche einer Simulations- und Demonstrationssoftware für drahtlose Sensornetze.
- Die entwickelte Visualisierungsbibliothek für OpenGL wurde im Hinblick auf die Simulation von drahtlosen Sensornetzen entworfen und für den Einsatz unter C# angepasst.

Thesen der Arbeit

- Die objektorientierte Strukturierung und der Einsatz des SceneGraphs vereinfacht die 3D-Grafik Programmierung deutlich.
- Die Einsatzfähigkeit der Visualisierungsbibliothek wurde an einer Testanwendung mit weitreichenden Funktionalitäten nachgewiesen.
- Die Weiterentwicklung der Visualisierungsbibliothek bietet großes Potential.

A Anlagen

A.1 Pflichtenheft Softwareoberfläche

Produktfunktionen

/PF10/	**Funktion**	MDI Konzept
	Beschreibung	Die Softwareoberfläche soll als "Multiple Document Interface" in Erscheinung treten. Das heißt, dass verschiedene Inhalte in einem Hauptcontainer dargestellt werden sollen.
/PF20/	**Funktion**	Erweiterung durch *WeifenLuo.WinFormsUI.Docking.dll*
	Beschreibung	Jeder Aufgabenbereich muss in einem separaten Fenster, dem "DockContent" dargestellt werden können. Die erweiterten Funktionalitäten werden durch die Bibliothek von *Weifen Luo* bereit gestellt.
/PF30/	**Funktion**	Speicher- und Ladefunktion der Oberflächenansicht
	Beschreibung	Das Speichern und Wiederherstellen der GUI soll mittels XML-Dateien geschehen, die persistent gespeichert werden und die Dateiendung LAYOUT tragen. Es müssen beliebig viele LAYOUT-Dateien über die Menüpunkte Preset Laden / Speichern erzeugt und zur Laufzeit geladen werden können.
/PF40/	**Funktion**	Ein und Ausblenden der Inhalte
	Beschreibung	Über den Menüpunkt „Fenster“ soll der Nutzer selbst entscheiden können, welche konkreten Inhalte angezeigt werden sollen und welche verborgen werden sollen.
/PF50/	**Funktion**	Batteriewerte, Lebensdauer
	Beschreibung	Es muss für jeden Netzwerkteilnehmer, der über eine endliche Energiekapazität verfügt, der Batteriewert in

		einem Diagramm angezeigt werden können. Neben der Einzelansicht soll es auch für das gesamte Netzwerk eine grafische Darstellung der Überlebensrate geben.
/PF60/	**Funktion**	Mehrsprachigkeit
	Beschreibung	Die Software soll mehrere Sprachen unterstützen, die über eine externe Klassenbibliothek *Languages.dll* eingebunden werden.
/PF70/	**Funktion**	Anzeige der Simulationsumgebung
	Beschreibung	Die Anzeige der Simulationsumgebung muss in 2D und in 3D möglich sein.
/PF80/	**Funktion**	3D-Darstellung
	Beschreibung	Die Visualisierung dreidimensionaler Inhalte erfolgt über die Einbindung der Klassenbibliothek *SharpOGL_AS.dll* Alle 3D-Komponenten müssen sich deaktivieren lassen, um die Geschwindigkeit der Anwendung nicht zu beeinträchtigen.
/PF90/	**Funktion**	Steuerung der Simulation
	Beschreibung	Die Simulation muss sich direkt, mit nur einem Mausklick, steuern lassen. Dazu gehört das Starten, Pausieren, ein Einzelschrittbetrieb und das Beenden einer Simulation.

A.2 Die MIT Lizenz

A.3 Implementierung DockPanel Bibliothek

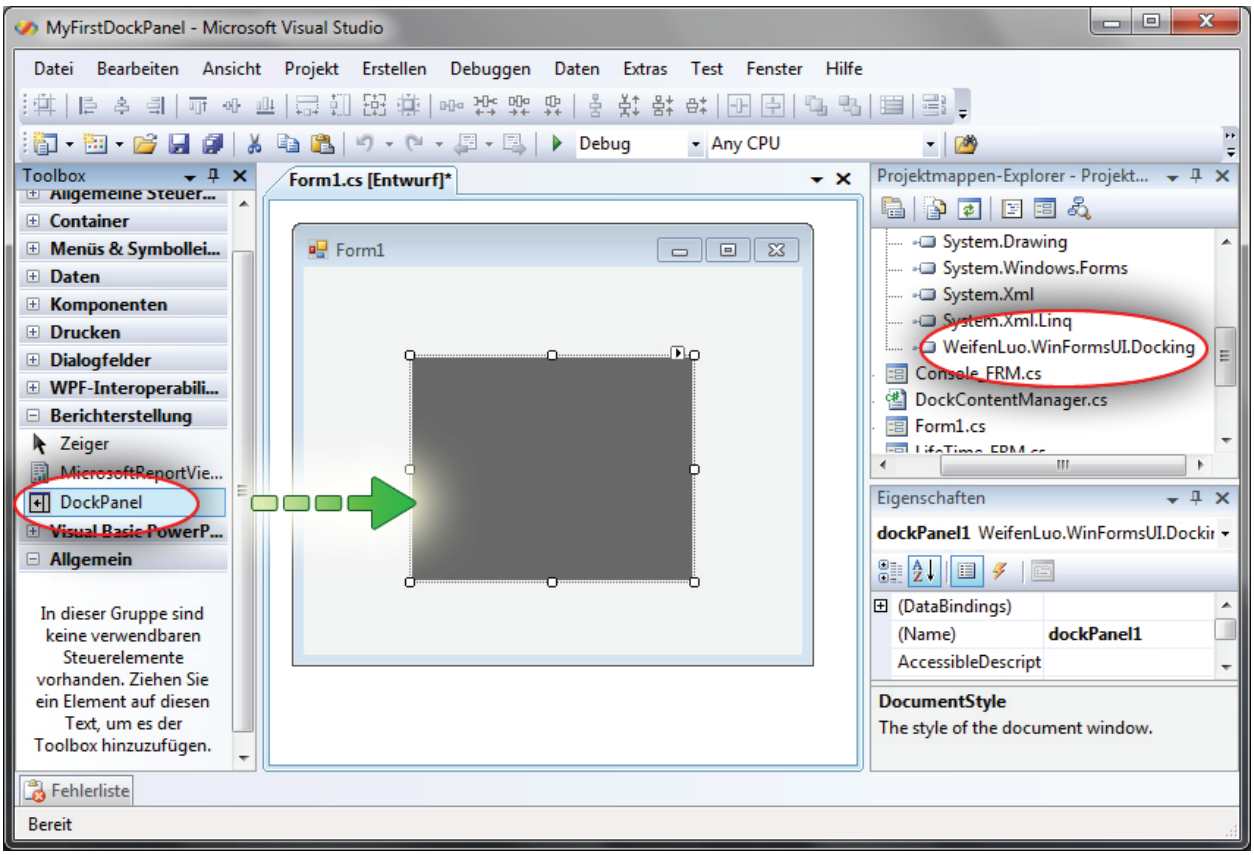


Abbildung A-1 Implementierung DockPanel

A.4 OpenGL Rendermodus

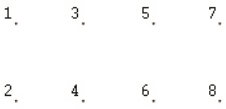
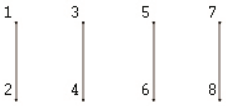
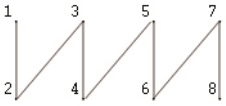
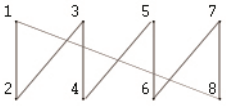
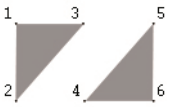
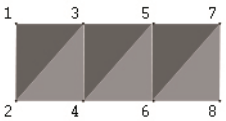
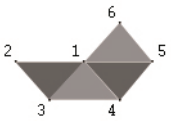
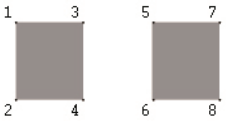
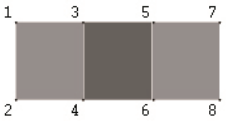
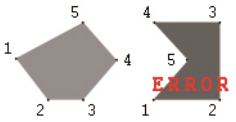
Rendermodus	Ergebnis
GL_POINTS	
GL_LINES	
GL_LINE_STRIP	
GL_LINE_LOOP	
GL_TRIANGLES	
GL_TRIANGLE_STRIP	
GL_TRIANGLE_FAN	
GL_QUADS	
GL_QUAD_STRIP	
GL_POLYGON	

Tabelle A-1 OpenGL Rendermodus

Originalquelle zur Tabelle A-1: [ARB 08] S. 44

A.5 Funktion für Objektselektion

```
1    private uint selectObject()
2    {
3        // speicehrt die Ergebnisse der Selektionsprüfung
4        uint[] buffer = new uint[256];
5        // speichert die aktuelle Ansicht (den viewport)
6        int[] view = new int[4];
7        // speichert die Anzahl der getroffenen Objekte
8        int hitsCount;
9        // hit = Objektname, zValue = Tiefen-Informationen des Objektes
10       UInt64 hit, zValue;
11
12       // den aktuellen Viewport holen
13       gl.GetInteger_v(GL_DEF.GL_VIEWPORT, ref view);
14       // den buffer zuordnen
15       gl.SelectBuffer(256, ref buffer);
16
17       // in den Selektions-Modus wechseln, Namensstapel initialisieren
18       gl.RenderMode(GL_DEF.GL_SELECT);
19       gl.InitNames();
20       gl.PushName(0);
21
22       // in den Projektionsmodus wechseln, Matrix initialisieren
23       gl.MatrixMode(GL_DEF.GL_PROJECTION);
24       gl.PushMatrix();
25       gl.LoadIdentity();
26
27       gl.glu.PickMatrix( mouse.X, (view[3] - mouse.Y), 1.0, 1.0, view);
28       gl.glu.Perspective(45.0f, (float)scene_CTRL.ScenePanel.Width /
                            (float)scene_CTRL.ScenePanel.Height,
                            0.1f, 500.0f);
29
30       // --- hier muss zwischendurch die Szene gerendert werden
31       renderScene();
32
33       // Wieder in den Projektionsmodus wechseln
34       gl.MatrixMode(GL_DEF.GL_PROJECTION);
35       gl.PopMatrix();
36
37       // jetzt werden die getroffenen Objekte gesammelt
38       hitsCount = gl.RenderMode(GL_DEF.GL_RENDER);
39
40       hit = 256;
41       zValue = Int64.MaxValue;
42
43       for(int i = 0; i < hitsCount; i++ )
44           if ( buffer[(i*4) + 1] < zValue )
45           {
46               hit = buffer[(i*4) + 3];
47               zValue = buffer[(i*4) + 1];
48           }
49
50       // Ergebnis ausgeben
51       if(hit == 256)
52           return 0;
53       else
54           return (uint)hit;
55   }
```

Quellcode A-1 Objektselektion

Originalquelle zum modifizierten C# Quellcode A-1: [DGL 09]

Quellenverzeichnis

Internetquellen

[Schubert 09] Prof. Dr. Schubert, Wilfried: Grundlagen der Softwaretechnik
URL: https://www.htwm.de/~wschub/intranet/ss09/Fach_SWT/Fach_SWT_VL08_klein.pdf
Zugriffszeit: 01.01.2010

[SG 09] Wikipedia - Die freie Enzyklopädie: Szenengraph
URL: http://de.wikipedia.org/wiki/Szenengraph
Zugriffszeit: 21.12.2009

[CsGL 03] CsGL - C# graphics library
URL: http://csgl.sourceforge.net/
Zugriffszeit: 28.12.2009

[TK 09] The Open Toolkit Library
URL: http://www.opentk.net/
Zugriffszeit: 28.12.2009

[OGL 09] OpenGL - The Industry's Foundation for High Performance Graphics
URL: http://www.opengl.org/
Zugriffszeit: 14.01.2010

[C4D 09] Cinema4D - MoGraph2 Samples
URL: http://www.maxon.net/en/products/new/mograph-2/modynamics.html
URL
Video: http://maxonpodcast.de/video/products/new/mograph_2/modynamics/falling.mov
Zugriffszeit: 23.12.2009

[DGL 09] DGL Wiki - freies OpenGL-Wissen für Alle
URL: http://wiki.delphigl.com/index.php/Tutorial_Selection
Zugriffszeit: 10.01.2009

verwendete Software

[Avalon 09] CodePlex - Open Source Project Hosting
URL: http://www.codeplex.com/Wikipage?ProjectName=AvalonDock
Zugriffszeit: 12.04.2009

[DC 09] CodeGuru - A Docking Panels Library in .NET 2.0
URL: http://www.codeguru.com/csharp/csharp/cs_controls/custom/article.php/c12875/
Zugriffszeit: 12.04.2009

[Luo 09] SourceForge - DockPanel Suite
URL: http://sourceforge.net/projects/dockpanelsuite/
Zugriffszeit: 12.04.2009

[Cinema4D] Cinema4D Release 10
URL: http://www.maxon.net/de/home.html
Zugriffszeit: 09.01.2010

Literaturquellen

[ISO9241 06] DIN EN ISO 9241-110:2006-08: Ergonomie der Mensch-System-Interaktion - Teil 110: Grundsätze der Dialoggestaltung (ISO 9241-110:2006);
Deutsche Fassung EN ISO 9241-110:2006, Beuth-Verlag, 2006

[Großmann 09] Großmann, Toni Dirk: Anforderungsanalyse und Systementwurf einer Simulationssoftware für drahtlose Sensornetze - 2009 - 100 S.
Mittweida, Hochschule Mittweida, Fakultät: Mathematik / Physik / Informatik, Diplomarbeit, 2009

[Vonhoegen 09] Vonhoegen, Helmut: Einstieg in XML Grundlagen, Praxis, Referenz - 5. Auflage -
Bonn: Galileo Press, 2009

[Huber 08] Huber, Thomas Claudius: Windows Presentation Foundation:

Das umfassende Handbuch - 1. Auflage -
Bonn: Galileo Press, 2008

[ARB 08] Shreiner, Dave, ...: OpenGL Programming Guide - 6. Auflage -
Boston: Pearson Education, Inc., 2008

[Apetri 08] Apetri, Marius: 3D-Grafik Programmierung - 2. Auflage -
Heidelberg: REDLINE GmbH, 2008

Erklärung zur selbstständigen Anfertigung der Arbeit

Hiermit erkläre ich, Adrian Singer, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Mittweida, Mittwoch, 30. Januar 2010

Bearbeitungsort, Datum	Unterschrift